Muhammad Usman Iftikhar

A Model-Based Approach to Engineer Self-Adaptive Systems with Guarantees

Dissertation presented in partial fulfillment of the requirements for the dual degree of Doctor of Computer Science from Linnaeus University and Doctor of Engineering Science (PhD): Computer Science from KU Leuven.

Supervisor(s):
Prof. Danny Weyns
Dr. Jesper Andersson

December 18, 2017

Muhammad Usman Iftikhar

A Model-Based Approach to Engineer Self-Adaptive Systems with Guarantees

Doctoral Thesis

Computer Science

2017

Linnaeus University

KU Leuven

A thesis for the dual degree of Doctor of Computer Science from Linnaeus University and Doctor of Engineering Science (PhD): Computer Science from KU Leuven.

A Model-Based Approach to Engineer Self-Adaptive Systems with Guarantees

Muhammad Usman Iftikhar
usman.iftikhar@lnu.se, muhammadusman.iftikhar@kuleuven.be

Linnaeus University
School of Computer Science, Physics and Mathematics
SE-351 95 Väjxö, Sweden, 2014
http://www.lnu.se/

KU Leuven
Faculty of Engineering Science, Department of Computer Science
Celestijnenlaan 200A B-3001 Heverlee (Belgium)
https://www.kuleuven.be

ISBN: 978-91-88761-04-0 (print), 978-91-88761-05-7 (pdf)

.

To my family and friends

Abstract

Modern software systems are increasingly characterized by uncertainties in the operating context and user requirements. These uncertainties are difficult to predict at design time. Achieving the quality goals of such systems depends on the ability of the software to deal with these uncertainties at runtime. A self-adaptive system employs a feedback loop to continuously monitor and adapt itself to achieve particular quality goals (i.e., adaptation goals) regardless of uncertainties. Current research applies formal techniques to provide guarantees for adaptation goals, typically using exhaustive verification techniques. Although these techniques offer strong guarantees for the goals, they suffer from well-known state explosion problem. In this thesis, we take a broader perspective and focus on two types of guarantees: (1) functional correctness of the feedback loop, and (2) guaranteeing the adaptation goals in an efficient manner. To that end, we present ActivFORMS (Active FORmal Models for Self-adaptation), a formally founded model-driven approach for engineering self-adaptive systems with guarantees. ActivFORMS achieves functional correctness by direct execution of formally verified models of the feedback loop using a reusable virtual machine. To efficiently provide guarantees for the adaptation goals with a required level of confidence, ActivFORMS applies statistical model checking at runtime. ActivFORMS supports on the fly changes of adaptation goals and updates of the verified feedback loop models that meet the changed goals. To demonstrate the applicability and effectiveness of the approach, we applied ActivFORMS in several domains: warehouse transportation, oceanic surveillance, tele assistance, and IoT building security monitoring.

Keywords: Self-adaptive software systems, MAPE-K feedback loop, Statistical model checking, Analytical methods

Acknowledgments

With this dissertation, I accomplish a significant milestone in my life, but it was not possible without the help of my family, teachers, and friends. I like to thank all the people who helped me during my studies.

My sincerest thanks to my supervisor and mentor, Professor Danny Weyns, for providing me a chance to pursue my dream. Without his encouragement and guidance, I wouldn't have been able to realise this dream.

I also want to thank my co-supervisor Jesper Andersson and examiner Prof. Welf Löwe, for providing me all the support to accomplish this milestone.

Special thanks to Jonas Lundberg, who has always believed in me and provided support whenever I needed any guidance and help.

Sincere thanks to all my colleagues and friends Nadeem Abbas, Sharafat Ali, Stepan Shevtsov, Didac Gil de la Iglesia, and Asim Raza, who supported me always during my study.

Special thanks to all my teachers, especially, Onaiza Maqbool, who not only learned me a lot on software engineering, but also guided me to become a better person.

My research collaborators, Prof. Radu Calinescu and Simos Gerasimou, Prof. Danny Hughes and Gowri R. Sankar, special thanks for providing support in my research.

Special thanks to all the jury members, especially, Prof. Flavio Oquendo, Romina Spalazzese, Patrizio Pelloccione, and Maruo Caporuscio, for providing me valuable comments.

Special thanks to all the faculty member of Linnaeus University and KU Leuven University, for supporting me with all the hard work.

I want to thank my mother Farzana Shamim who always believed in me. Without her prayers, I wouldn't be able to accomplish this achievement. After that, I want to thank my grandfather Muhammad Shafi, who always supported me in pursuing my studies and for every other aspect of my life. I want to thank my wife Adeela Usman for providing me moral support and well taken care of me and my son Muhammad Arsh Usman who was always supportive when I was studying. I am also grateful to my other family members, my sister Summaira Shamail and her family, my cousins and friends who have supported me along the way.

Last but not least, I want to thank Almighty Allah, who guided me all my life, providing me with countless blessings.

Växjö, Sweden
November 24, 2017

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

With the advance of technology, software has become an essential and integrated part of everyone's life. Modern computing systems consist of many hardware and software components working together to achieve particular goals. Examples are multi-robot systems that support rescue workers with difficult tasks in disaster areas, unmanned underwater vehicles that monitor pollution levels of maritime areas, and smart homes that exploit Internet of Things (IoT) technology to monitor the environment and control appliances for lighting, heating, doors, etc.

The increasing pervasiveness and mobility of modern software systems pose several challenges to software engineers. One of the critical challenges is the ability of the software system to handle a highly uncertain environment, with minimum human intervention. Software systems have to deal with many uncertainties during operation, such as dynamics in the availability of resources and services, changing operating conditions, and evolving user needs. Developing these systems is particularly challenging, since software engineers often do not have complete knowledge about the uncertainties at development time. Without any mitigation mechanism, the uncertainties can lead to undesired behavior of the software system, for example, the performance can degrade or the functionality can be compromised due to reliability issues [93, 109].

For a modern software system, it is almost indispensable to continuously configure and optimize itself at runtime when knowledge becomes available that is required to resolve the uncertainties. Self-adaptation is widely considered as an effective approach that enables a software system to deal autonomously with uncertainties and dynamics of the environment [48, 65, 133]. A self-adaptive system tracks uncertainties in its environment, reasons about the uncertainties, and adapts itself to maintain the system goals, or degrade gracefully when required [205].

Over the years, many adaptation techniques have been purposed to realize the adaptation behavior [84, 122, 215]. One of the principled techniques is called *architecture-based self-adaptation* [117]. In architecture-based self-adaptation, a self-adaptive system consists of two parts: a managed system and a managing system. The *managed system* provides the domain functionality to the user and is the subject of adaptation. The *managing system* adapts the managed system when required, providing the adaptation behavior. The managed system needs to be instrumented with *probes* and *effectors* that enable the managing system to monitor the managed system and its environment and adapt the managed system as needed.

A common approach to realize the managing system is using one or more *MAPE-K* feedback loops. A MAPE-K feedback loop consists of five distinct components: *Monitor* tracks uncertainties in the managed system and environment using probes, *Analyzer* performs an analysis of the monitored data and estimates available adaptation options and determines whether adaptation is required, *Planner* picks an adaptation option that realises a set of adaptation goals (adaptation goals are typically quality goals [208, 212]) and creates a plan, i.e., a set of actions that are required to adapt the managed system, and *Executor* executes the plan using the effectors provided by the managed system. *Knowledge* provides a repository that enables the MAPE components to share data. The shared repository is used to store data about the managed system, the environment, adaptation goals, and the MAPE components themselves.

The notion of uncertainty has been studied extensively in a variety of fields, see for example [25, 29, 146]. In the context of self-adaptation, uncertainty has only been subject of research recently [69, 141, 154]. In our research, we limit our scope to parametric uncertainty. Parametric uncertainty is characterised by a lack of knowledge at design time about values of parameters of the system and its environment. Parametric uncertainty is a form of epistemic uncertainty [154, 184]. The exact values of the uncertainty parameters will only be available at runtime when the software system operates in the environment. It is important to note that the parameter values are subject to change over time. An example of a parametric uncertainty in the case of an IoT application is the interference in the links of the wireless network that connects motes. The concrete levels of interference will only be known at runtime when the motes send messages to each other. Note that the rate of interference for different links can change over time, due to many reasons such as the presence of users, changing weather conditions, etc. Hence a self-adaptation solution is required that monitors the uncertainty parameters at runtime and adapts the system accordingly to deal with changing conditions. Other types of uncertainty, such as structural uncertainty are out of scope of our research.

Although self-adaptive systems can deal with uncertainties autonomously, providing guarantees that the system goals are achieved is still a challenging problem [40, 135]. Especially, providing guarantees for systems with strict goals is crucial since these systems rely on the correct operation of the software to meet their goals.

1.1 Problem Definition

Despite the advances of research on self-adaptation, one of the critical aspects that remains to be tackled is the provision of the guarantees that the self-adaptive system achieves its goals [7]. Providing such guarantees is challenging since software systems are exposed to various types of uncertainty that can appear at anytime. An important objective of our research is to provide guarantees for the goals of self-adaptive systems in an efficient way. The following research questions are investigated in the scope of this thesis:

RQ1 How to provide functional correctness of the feedback loop?

RQ2 How to efficiently provide guarantees for the adaptation goals of the system?

RQ3 How to support on-the-fly changes of adaptation goals?

With RQ1, we aim to achieve functional correctness of the feedback loop components, i.e., MAPE-K components. With RQ2, we aim to guarantee the adaptation goals of a software system in an efficient way, such that the approach is applicable in practice. With RQ3, we aim to enable adaptation goals to change on-the-fly, which typically requires updates of the feedback loop components at runtime to support the changing goals.

Recent research in self-adaptation suggests using formal methods for providing guarantees for the goals of self-adaptive systems [38, 214]. Formal methods provide the means to rigorously specify mathematical models of a software system and verify its behavior with respect to required properties [104].

Existing approaches mainly focus on using formal methods for two types of guarantees: 1) guarantees for correct behavior of adaptive systems, and 2) guarantees for the adaptation goals of the system. A pioneering approach that focusses on guarantees for the correct behavior of adaptive systems is described in [112]. This approach uses formal models (Petri nets) to specify and verify the adaptive and non-adaptive behavior of a system. After that, the formal models are automatically translated into executable programs. In this approach, model-based testing techniques are applied to check the conformance between the models and the executable programs.

A prominent approach that mainly focuses on ensuring adaptation goals is Runtime Quantitative Verification (RQV) [37]. RQV typically uses a parameterized Markov model of the system and its environment. There are two types of parameters in the model: first, variabilities of the system that determine the adaptation options, and second uncertainties typically in the environment. The parameters that represent uncertainties are updated at runtime via monitoring. A model-checking tool then verifies the updated model to evaluate the quality properties for each adaptation option that needs to be checked. After that, an adaptation option is selected that fulfils the required adaptation goals of the system. A detailed overview of the state of the art is provided in sections 4.8 and 6.6.

In general, the primary focus of state of the art is on ensuring the required adaptation goals that are subject of adaptation. Guaranteeing the functional correctness of the feedback loop is often ignored. For example, if the Monitor component uses a learning function to track changes in uncertainty, then does that function work correctly? Is the Analyzer able to perform the analysis correctly? Will the plan created by the Planner fulfill system goals? Is the plan executed in the correct order by the Executor? Guaranteeing such properties is crucial for the feedback loop to work correctly, which is the goal of RQ1.

Existing approaches that focus on ensuring the required adaptation goals typically rely on the exhaustive verification, which suffers from the well-known state explosion problem [53]. Exhaustive verification is known to be computationally very demanding and may not be applicable in resource-constrained systems, for

example, embedded systems that have low memory and processing power. Therefore, with RQ2, we study how to provide guarantees for the adaptation goals in an efficient way.

Furthermore, existing approaches do not pay much attention on change adaptation goals on-the-fly. However, changing adaptation goals on-the-fly is considered a fundamental feature of a self-adaptive system [122, 165, 175]. With RQ3, we study how to enable on-the-fly changes of adaptation goals.

1.2 Scope of the Research

Any research endeavour has a particular scope. We divided the scope of the research presented in this dissertation in three categories: 1) the class of problems that our research can solve by applying self-adaptation, 2) the type of adaptation that our research targets, and 3) the assumptions that are made that put restrictions on the scope of this research.

1.2.1 Class of problems

Our research targets adaptation problems for different types of quality properties of systems. Quality properties are non-functional properties of the system that indicate *how* well the system is able to deliver its intended functionality. We focus in particular on performance, reliability, and efficiency requirements. We consider in particular requirements to keep a value above or below a given threshold, or requirements that optimize a particular value. In addition, we also support the evolution of quality properties to deal with changed requirements or adding a new quality property at runtime to deal with new user requirements. Other types of qualities, such as security or interoperability, are out of the scope of this research.

1.2.2 Types of adaptation

We limit our research to the systems where a central feedback loop can monitor and adapt the system. In the case, the software system is distributed, we focus on systems where the Analyzer and Planner components of the feedback loop are running on the same node. Other types of adaptation, where multiple feedback loops are required (e.g. [44]), such as in systems-of-systems (e.g. [148]), are out of the scope of this research. This also includes adaptation solutions that are based on the principles of self-organisation [86].

1.2.3 Assumptions

Our research relies on the following assumptions:

- We assume that the managed system already exists. We also assume that the managed system provides the necessary infrastructure to support self-adaptation, including probes and effectors.
- We consider systems for which adaptation is event- or time-triggered. Furthermore, we target systems in which dynamics of the environment is slower than the time required by the feedback loop to adapt the system.

- We consider systems with a finite set of adaptation options among which the adaptation logic makes a selection to be executed. Discretising continuous domains or applying heuristics to select adaptation options in a large space can complement our work, but are out of scope of this research.

Finally, in this research, we follow the paradigm of architecture-based self-adaptation that uses MAPE-K components to realize the feedback loop.

1.3 Scientific Approach

To conduct our research and evaluate the research results, we used analytical methods and empirical methods respectively. Analytical methods are used to model and verify the behavior of software systems rigorously. We used three specific analytical methods in our research: model checking, runtime simulation, and statistical model checking. A detailed description of these analytical methods is provided in chapter 2. Empirical methods are used to collect evidence for the proposed solutions. We used *systematic empirical inquiry* as an empirical method in our research [81]. A systematic empirical inquiry follows a rigorous approach to evaluate a particular solution by defining objectives, setting up an experiment, collecting and analyzing data to presenting findings in a systematic manner.

We conducted our research in three phases in which both analytical methods and systematic empirical inquiry are used to answer the research questions. Figure 1.1 presents an overview of research phases with the research goals and applied methods.

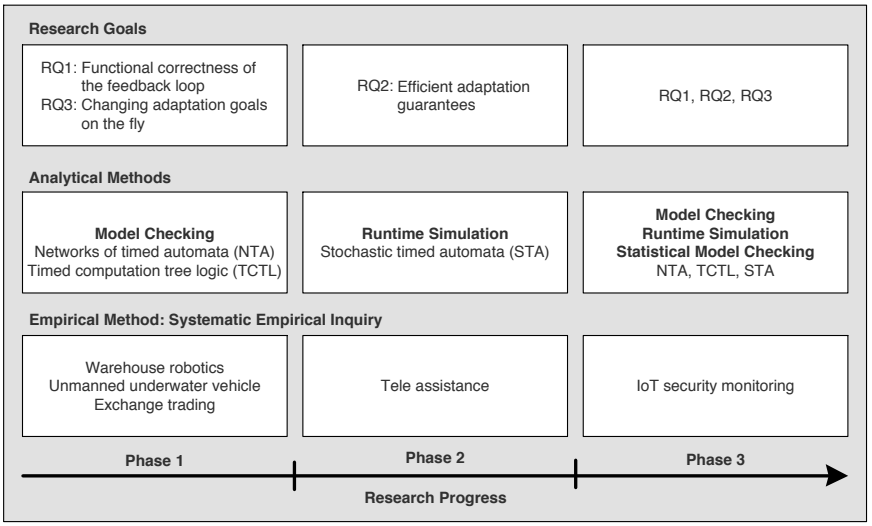


Figure 1.1: Overview of the scientific approaches used to conduct the study

In the first phase, we focused on RQ1 (functional correctness of the feedback loop) and RQ3 (changing adaptation goals on the fly). We applied model checking

as the analytical method to provide functional correctness of the feedback loop. Concretely, we used networks of timed automata (NTA) to specify feedback loop models and timed computation tree logic (TCTL) to specify correctness properties and verify them. We evaluated the research results using a systematic empirical inquiry in three domains: a simple warehouse robotic system, an unmanned underwater vehicle application, and a exchange trading system.

In the second phase, we focused on research question RQ2 (efficiently providing guarantees about adaptation goals). We applied runtime simulation as the analytical method using stochastic timed automata to provide guarantees for the adaptation goals. We evaluated the research results using a systematic empirical inquiry for a tele assistance system.

The first and second phase result in an initial version of ActivFORMS. In the third and final phase our research, we developed a mature and integrated version of ActivFORMS that deals with the three research questions. We combined different analytical methods, including model checking, runtime simulation, and statistical model checking. We evaluated the integrated solution using a systematic empirical inquiry for a real-world IoT security monitoring application deployed at KU Leuven. We compared our solution with a state of the art approach that uses runtime quantitative verification (RQV).

1.4 Overview of ActivFORMS Approach

This thesis contributes ActivFORMS (Active Formal Model for Self-Adaptation), a model-based integrated approach for engineering self-adaptive systems. ActivFORMS provides guarantees for: 1) functional correctness of the feedback loop, 2) efficient guarantees about the adaptation goals, and 3) support for on-the-fly changing adaptation goals.

Figure 1.2 shows a high-level overview of ActivFORMS approach that consists of four main stages of the software lifecycle of an adaptive system: 1) design, 2) deployment, 3) runtime adaptation, and 4) evolution of the feedback loop [205]. We discuss each stage briefly.

Stage 1: model and verify feedback loop. In stage 1, formal models for the feedback loop are specified and verified. ActivFORMS provides a set of reusable MAPE templates that support the design of feedback loops. For modeling and verification, we use Uppaal tool suite.

Stage 2: deploy feedback loop model with virtual machine. In stage 2, the verified formal model of the feedback loop are deployed together with the ActivFORMS virtual machine. This virtual machine executes the formal model of the feedback loop to realise self-adaptation. Stages 1 and 2 realise the functional correctness of the feedback loop.

Stage 3, verify adaptation goals and adapt. In stage 3, the feedback loop monitors the managed system and its environment, uses runtime simulation and statistical model checking to realise the adaptation goals by adapting the managed system. Stage 3 realises the adaptation goals in an efficient manner.

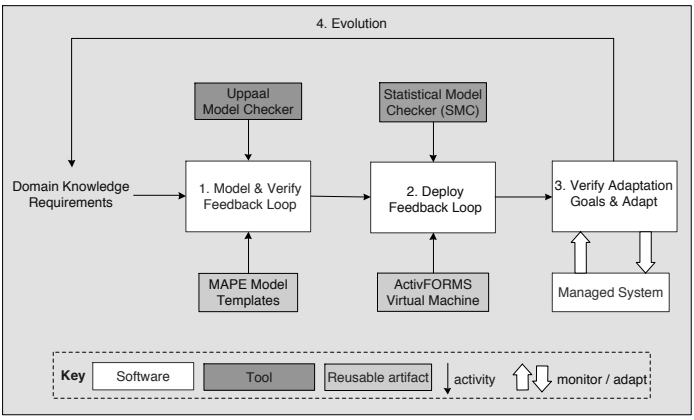


Figure 1.2: Overview of ActivFORMS approach [205]

Stage 4, evolve adaptation goals and feedback loop model. Stage 4 supports for on-the-fly change of adaptation goals and the feedback loop model. Stage 4 enables ActivFORMS to deal with new or changing adaptation goals, or to improve the functions of the MAPE components.

It is important to note that on-the-fly updates supported by ActivFORMS are restricted as follows:

1. Updates of the feedback loop model that require an update of the managed system, the probes or effectors, is out of scope of our research.
2. Support for state updates after the new model is loaded is limited to restoring the state of the same variables (in the old and the new model) or initialising new variables (in the new model).
3. The execution of input messages of the monitor that are buffered during an update of the feedback loop model do not generate any effects others as their intended purpose.

1.5 Contributions

This thesis contributes a novel approach for engineering self-adaptive systems called *ActivFORMS*. ActivFORMS adds the following contributions to the state-of-the-art:

1. An innovative approach to provide functional correctness of the feedback loop. Functional correctness is achieved using a reusable virtual machine that directly executes the formally verified model of the feedback loop. Hence, the approach ensures the design time guarantees at runtime. ActivFORMS comes with a set of reusable templates that help designers to create MAPE-K components [40, 107, 109, 205].

2. An efficient approach to provide guarantees for the adaptation goals. In particular, the approach employs statistical model checking at runtime to efficiently estimate quality properties of the system. Compared to exhaustive model checking, statistical model checking provides verification results with a require level of accuracy and confidence. However, the engineer can set the level of accuracy and confidence in terms of the available resources [205, 209].
3. A novel approach for changing adaptation goals on-the-fly, together with changing models of the feedback loop to support the changing goals. The approach applies the principles of quiescence [123] to runtime models to enable updating the goals and models of the feedback loop safely during operation [109, 205].

1.6 Overview of the Thesis

We conclude this introduction with an overview of the thesis. Figure 1.3 gives a schematic overview of the chapters with technical contributions that are linked to the research questions. Chapters 3 to 6 and Appendix A and B are copies of published papers.

In chapter 2, we provide an introduction of the basics of self-adaptive systems, we briefly discuss related approaches, and we introduce the analytical and empirical methods we use in our research.

In chapter 3, we introduce the initial version of ActivFORMS approach that uses a virtual machine to execute formal models during operation. The focus of this chapter is on functional correctness of the feedback loop (RQ1) and changing adaptation goals on-the-fly (RQ3). We evaluated ActivFORMS with a simple robotic warehouse transportation system. This chapter is a copy of [109].

In chapter 4, we integrated ActivFORMS with RQV and a dynamic assurance case (i.e., an assurance case that is dynamically updated after each adaptation of the system). The resulting engineering approach, called *ENTRUST*, helps engineers designing trustworthy self-adaptive software systems. We evaluated ENTRUST to an unmanned underwater vehicle (UUV) system and a foreign exchange trading (FX) system. This chapter is a copy of [40].

In chapter 5, we use runtime simulation to evaluate the expected qualities of the different adaptation options in an efficient way (RQ3). This enables the feedback loop to select the best adaptation option. We evaluated the simulation-based approach to a tele assistance system. This chapter is a copy of [209].

In chapter 6, we present the integrated ActivFORMS approach, a formally founded model-driven approach for engineering self-adaptive systems (RQ1, RQ2, RQ3). The approach integrates: 1) functional correctness of the feedback loop by direct execution of formally verified models of the feedback loop using a reusable virtual machine (RQ1), 2) efficient guarantees for the adaption goals with a required level of confidence using statistical model checking techniques at runtime (RQ2), and 3) support for changing adaptation goals on the fly and updating of verified models of the feedback loop that meet the new goals (RQ3). We evaluated

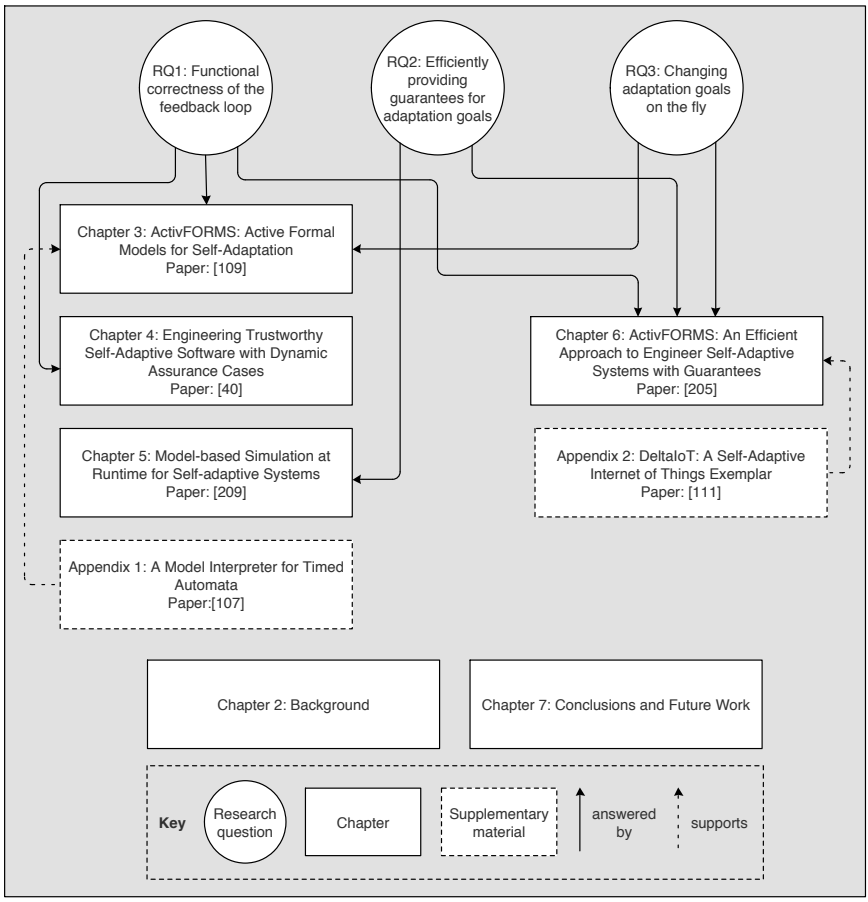


Figure 1.3: Overview of the thesis with research questions and their answers

the approach to a real-world deployed IoT building security monitoring application. This chapter is a copy of [205].

Finally, we draw conclusions in chapter 7. We summarise the contributions of our research, we report lessons learned, and outline interesting paths for future research.

The dissertation includes two appendixes. Appendix A provides a technical description of the virtual machine used in ActivFORMS. Appendix B, presents DeltaIoT, an exemplar that enables researchers to evaluate and compare new methods, techniques and tools for self-adaptation in IoT. We used DeltaIoT in chapter 6 to evaluate the ActivFORMS approach. Appendix A and B are a copy of [107] and [111] respectively.

Chapter 2

Research Background

In this chapter, we discuss the basic concepts and methods that underlie our research. We start with explaining the principles of self-adaptation and highlight two references models. Then, we discuss a number of state of the art approaches that are closely related to our work. After that, we explain the analytical methods that we used in our research, i.e., model checking, runtime simulation, and statistical model checking. The chapter conclude with a brief discussion of systematic empirical inquiry, the empirical approach we used to validate the research results.

2.1 Self-Adaptation

Self-adaptation enables a software system to deal with uncertainties by reconfiguring its structure or adjust its behavior during operation. Examples of uncertainties are dynamic availability of resources and evolving user requirements. In this research, we focus on architecture-based self-adaptation. Architecture-based self-adaptation separates the concerns of the *managed system* that is subject to adaptation and the the concerns of the *managing system* that contains the adaptation logic. The managing system realizes a feedback loop that monitors and adapts the managed system to achieve the adaptation goals. Hereinafter, we discuss key reference models that are used in our research to realize the managing system.

2.1.1 3-Layer Architecure Model

In 2007, Kramer and Magee proposed a reference model to realise self-adaptation [122] that is inspired by the three-layer architecture for robotic systems proposed by Gat [85]. The three layers of the proposed reference model are: component control, change management, and goal management.

Figure 2.1 shows an overview of the three layer reference model. The bottom layer Component Control consists of the software system that is subject of adaptation. This software system consists of interconnected components that may be distributed over a network of computing nodes. The software system is equipped with sensors that report the current status of components to the upper layer and actuators that enable modification of the components, such as addition, deletion, and interconnection. The Change Management layer comprises a set of plans that can be used to respond to status changes of the bottom layer. An example of a status change is a failure of a component. Such a status change triggers an appropriate

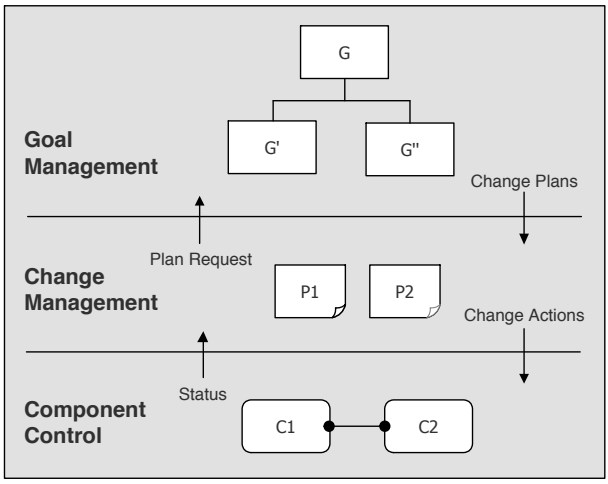


Figure 2.1: Overview of the 3-layer reference model [122]

plan of the change management layer, for example to create new component(s) or change interconnections. Finally, the Goal Management layer deals with status changes of the bottom layer that cannot be handled by the plans available at the change management layer. To that end, the goal management layer create new plans in response to the requests from the layers below, or alternatively in response to new or changing goals.

2.1.2 MAPE-K Reference Model

In 2003, IBM introduced the vision of autonomic computing that is inspired by our autonomic nervous system that governs our heart rate, body temperature, etc. [57]. Central to this vision is the so called MAPE-K reference model [117].

Figure 2.2 shows a schematic overview of a self-adaptive system with a MAPE-K feedback loop. A self-adaptive system is composed of a managed system that is subject of adaptation and a managing system that realises a MAPE-K feedback loop to adapt the managed system. The environment provides context to the managed system in which this system operates. The managed system and managing system are connected through sensors and actuators. The sensors provided by the managed system allow the managing system to sense the managed system, the actuators allow the managing system to adapt the managed system when necessary.

A MAPE-K feedback loop consists of five distinct components that define the following basic functions:

Monitor: The monitor component provides mechanisms to collect, aggregate, and filter runtime data collected from the sensors. This data allow resolving uncertainties.

Analyze: The analyze component analyses the data collected by the monitor to determine the need for adaptation; examples of analytical models are queuing models and Markov models.

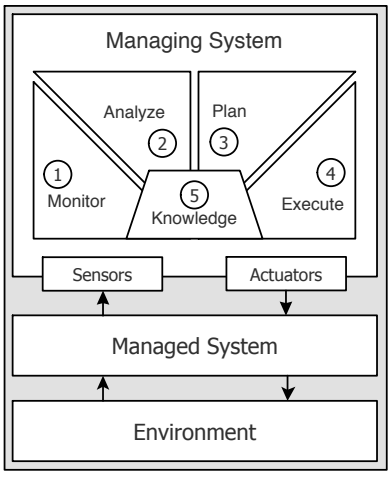


Figure 2.2: An overview of a self-adaptive system with MAPE-K reference model [57]

Plan: The plan component constructs a plan consisting of actions needed to adapt the managed system to achieve the adaptation goals.

Execute: The execute component executes the actions of the plan adapting the managed system through the actuators.

Knowledge: Knowledge is maintained in a repository that is shared among the MAPE components. Examples of knowledge are the current state of the managed system and environment, adaptation goals, analysis results, and plans.

2.2 Related Approaches

We discuss a number of key approaches for self-adaptation that are closely related to our research. Subsequently, we look at approaches that focus on functional correctness, quality guarantees, executable models, and on-the-fly updates of goals. A detailed discussion of the related work is included in the next chapters.

2.2.1 Functional correctness

A pioneer work that provides functional correctness of a self-adaptive system is presented in [112]. The work follows a model-based approach to the development of self-adaptive systems. A Petri net of the system is designed that models the adaptive and non-adaptive behavior of the system. Adaptation is then modeled as a transition of the system. The approach considers different types of adaptations (one-point adaptation, guided adaptation, and overlap adaptation). The Petri net model can then be verified. When the system is found correct it can be implemented using automatic model transformations or by implementing the system and checking conformance of the implementation and the model using model-based testing.

Recent work [43] uses model-checking techniques to determine the resilience of self-adaptive systems that embody the MAPE-K model. The approach injects malformed inputs from probes to quantify the impact that controller failures have on the target system. The approach comprises two phases. In the first phase, controller failures are identified by injecting invalid inputs from probes during the different operational stages of MAPE-K loop. In the second phase, traces of controller failures are aggregated into discrete-time Markov models. After that the resilience of the system is quantified by verifying resilience properties expressed in probabilistic computation tree logic (PCTL) on the synthesized Markov model.

Compare to existing work, our research focuses on fine-grained functional correctness of MAPE feedback loops. At design time, we use model checking to verify the correctness of MAPE feedback loop components and their interactions. At runtime the formal model of the feedback loop is directly executed using a reusable virtual machine. The correctness of the virtual machine has been tested extensively. As such the model execution of the MAPE feedback loop model ensures the design-time guarantees at runtime.

2.2.2 Quality guarantees

One of the popular approaches to provide guarantees for quality properties is runtime quantitative verification. Runtime quantitative verification (RQV) is a technique based on mathematics to analyze the performance and reliability of software systems that exhibit stochastic behavior [38].

RQV uses a parameterized model of a system, for example in the form of a discrete-time Markov chain (DTMC) or a continuous-time Markov chain (CTMC). The parameters in the model represent uncertainties that are updated at runtime with actual values monitored from the system and the environment. The up-to-date model of the system is analyzed at runtime based on the quality requirements that are expressed formally in temporal logics. Examples of properties that can be analyzed include for example probability requirements, e.g., probability of packet loss within a network over a specific time period, and performance requirements, e.g., the expected latency of a service for a service-based system under a given workload.

A recent approach that combines runtime quantitative verification with sensitivity analysis to realise self-adaptation is presented in [73]. The authors propose a mathematical framework for run-time efficient probabilistic model checking. The approach follows a two step process: pre-computation at design time and verification at run time. The pre-computation step takes as input: (1) a DTMC model of the system augmented with rewards, (2) a set of variable labels that are model parameters whose values are determined at runtime, and (3) the desired system requirements expressed in reward based PCTL properties. The pre-computation step produces a partially evaluated set of symbolic expressions that represent verification conditions that can be efficiently evaluated at run time (as soon as changes occur in the environment). The verification step evaluates the symbolic expressions at runtime by binding concrete values to the variable labels. The proposed

approach supports sensitivity analysis, i.e. reasoning about the effects of changes. The results of such analysis can be used to select effective adaptation strategies.

RQV is an effective approach to provide guarantees for quality properties at runtime. However, the approach is based on exhaustive verification, which suffers from the well known state-explosion problem [198]. In contrast, the statistical approach we use in our research allows making a tradeoff between the available resources and verification time and the required accuracy.

2.2.3 Executable Models

One of the pioneering approaches that uses executable models to realise self-adaptation is presented in [200]. EUREMA (ExecUtable RuntimE MegAmodels) follows a model-driven approach for engineering self-adaptive systems. EUREMA provides a domain-specific modeling language (megamodels) for designing feedback loops, knowledge that is represented as runtime models of the managed system, and their feedback loop interactions. The megamodels are kept alive at runtime and directly executed by an interpreter. The megamodels can be adapted on the fly to support evolution. EUREMA supports multiple feedback loops to handle multiple concerns such as self-repair or self-optimization. Furthermore, EUREMA supports online monitoring and exporting snapshots of runtime megamodels to support engineers in analyzing the feedback loop.

EUREMA is clearly related to our work; however, the current version of the approach lacks a formal underpinning. On the other hand, EUREMA offers a domain specific modeling language to specify the feedback loop components. This is one of our long term goals as discussed in Chapter 7.

2.2.4 Changing goals

One of the few works that supports changing goals on-the-fly is presented in [147]. The proposed approach presents a generic methodology to dynamically update a controller of a system. This way, the approach is able to satisfy a new specification that based on changes of environment assumptions, requirements, and interfaces.

The proposed approach allows specifying correctness criteria for dynamic updates of a controller, including: when an updated controller can take control of the system, the reconfiguration of the environment while satisfying the old specification, and when to start satisfying the new specification. Once the criteria are specified, the approach automatically synthesis a controller that can perform a safe update, i.e., it guides the system to a state in which the update of the controller can safely start (while ensuring that the controller update will eventually occur).

Compared to our work, the proposed approach is based on hot-swap of the controllers where a new controller takes control of the system while satisfying the old specification as well. On the other hand, ActivFORMS first makes sure that the system is in a quiescence state [123] before an update, i.e., a state where no adaptation is active so that the old feedback loop models can be safely replaced with the new models. If an event happens that triggers the feedback loop while updating the feedback loop, these actions are queued and executed after the update.

2.3 Analytical Methods

Analytical methods provide means to model and verify the behavior of software systems rigorously. In this section, we describe the analytical methods that are used in our research. These analytical methods are model checking, runtime simulation, and statistical model checking.

2.3.1 Model Checking

Model checking is an analytical method that uses brute-force techniques to analyze whether a mathematical model of the system satisfies a given set of properties. Model checking explores all possible system scenarios in a systematic manner in order to determine whether the system model truly satisfies the given properties.

There are many mathematical approaches to model software systems. For example, Petri nets, Z language, etc. In our research, we used timed automata to model behaviors of the feedback loop. Concretely, we used networks of timed automata that allow modeling individual components of the MAPE loop as well as their interactions.

2.3.1.1 Timed Automata

Timed automata (TA) enable modeling and verifing the behavior of time-dependent systems. A timed automaton (or behaviour) is a finite state machine that is equipped with a set of of real-valued clocks. All clocks progress synchronously at one rate. The clock values can be inspected and reset. After a reset, a clock starts increasing its value again as time progresses [11].

In our research, we use the Uppaal suite that allows to define a network of timed automata model where an automaton can synchronize with other automaton(s) through channels. For example, using a channel x a sender $x!$ can synchronize with a receiver $x?$. Uppaal supports two types of channels: binary and broadcast. A binary channel allows a sending automaton to synchronize with only one receiver; the sender will be blocked in the absence of a receiver. A broadcast channel sends the signal to all available receivers; the sender will continue if there is no receiver.

Only one state per automaton can be active at a time, called *control* or *active* state. Uppaal offers a subset of a C-like language to define custom functions and data types, variables, arrays, and structures. For a network of automate, the state of the system is defined by the active states of all automata, the clock values, and the values of all the data variables.

There are two types of transitions between states: 1) *action transition*, which is further refined into internal transition and synchronization transition, and 2) delay transition. An automata may make a state transition separately, called *internal transition*, or it may synchronise with another automata through channels, called *synchronization transition*. If no action transition can be taken and time can progress, a *delay transition* can be taken. If no action transition can be taken and time cannot progress, then a *timelock* situation occurs, which means no further transition can be taken resulting in deadlock.

A timed automaton can be formally defined as a tuple $TA = (L, l_0, C, A, E, I)$:

L is a finite set of locations;

$l_0 \in L$ is the initial location in an automaton;

C is a finite set of clocks;

A is a finite set of actions–co-actions (synchronization transition) and the internal transition action;

$E \subseteq L \times A \times B(C) \times 2^C \times L$ is a set of edges connecting locations with an action, a guard and a set of clocks to be reset;

$I : L \rightarrow B(C)$ assigns invariants to the locations;

where $B(C)$ represents clock constraints over edges and locations [16].

An edge can be labeled with: a *guard* that expresses a condition to take a transition, an *update* function that defines an action to be taken when a transition is made (e.g, reset a clock), and a *synchronize* label that allows the automaton to synchronize with other automata on a complementary action. A location can be equipped with an *invariant* that constrains the time that can be spent in that location. The location must be left before the invariant becomes invalid.

Uppaal allows to define a location as a committed location, i.e., time cannot progress (no delay transition enabled) until any automaton is in a committed state. Committed states have priority and a next transition must use an outgoing edge from one of the committed locations.

2.3.1.2 Timed Computation Tree Logic (TCTL)

The goal of model checking is to verify the correctness of the system model against a set of verification properties. For this purpose, Uppaal uses a subset of timed computation tree logic (TCTL) properties. A verification property can be expresses as a *state formulae* or a *path formulae*. A state formulae is an expression over variables and automata states. A path formulae quantifies over paths (or traces) of the model; path formulae include reachability, safety, and liveness properties [16].

Reachability. A reachability property allows to verify that given a state expression ϕ is reachable by any path that the system takes. A reachability property in Uppaal is expressed as $E <> \phi$.

Safety. A safety property allows to verify that something bad will never happen. In other words, a given state expression ϕ should be satisfied along all the paths that system takes. A safety property is expressed as $A[]\phi$

Liveness. A liveness property enables to verify that something will eventually happen, i.e., if a state expression ϕ is satisfied then eventually state expression ψ will also be satisfied. A liveness property is expressed as $\phi \rightarrow \psi$.

2.3.2 Statistical Model Checking

Statistical model checking (SMC) has been put forward as an efficient alternative to traditional model checking techniques that are based on exhaustive verification [129]. SMC uses simulation and statistical techniques to decide whether the system model satisfies a given property with some degree of confidence [51, 219].

In our research, we used Uppaal-SMC, a statistical model checking tool supported by Uppaal that enhances networks of timed automata with stochastic semantics [59].

We used SMC at runtime for probability estimation of a given quality model. An example of probability estimation is the probability that packets get loss during a certain period in an Internet of Things application. For probability estimation, Uppaal-SMC uses Monte Carlo simulation [144].

A probability estimation query is formulated as:

$$p = Pr[bound](<> \psi)$$

The query produces a probability estimation p for an expression ψ with an approximation interval $[p - \varepsilon, p + \varepsilon]$ and confidence $(1 - \alpha)$ in a given *time bound*. The number of simulations required to reach the desired approximation interval and confidence level are automatically determined during the execution of the estimation query. The value of ε and α can be dynamically changed to tradeoff the required accuracy of the estimation with the available resources (e.g., computation power, memory, and adaptation time, etc).

2.3.3 Runtime Simulation

Statistical model checking provides a solution for checking probabilistic properties. For reward-based properties, our approach relies on runtime simulation. Runtime simulation provides an estimation of reward-based properties, such as cost associated to service invocations under a given work load, energy consumption of motes for a given time period in an Internet of Things application, etc.

For runtime simulation, we use the Uppaal-SMC tool suite that allows to simulate the model of the system specified as network of timed automata with stochastic semantics. In Uppaal-SMCA, a simulation query is defined as:

$$simulate N[<= bound]\{E1, ..., Ek\}$$

where N is the number of simulation runs to be performed for a given *time bound*, and $E1, ..., Ek$ is the list of expressions that will be monitored during simulation.

Similar to statistical model checking, the number of simulations N and the time bound *bound* can be updated dynamically to adjust the required accuracy of the results allowing a tradeoff with the available resources to perform the simulation.

A simulation query can only simulate the system for a given number of runs and time bound. In contrasts to SMC, when a simulation query is executed, the number of simulations required for the required accuracy of the results needs to be determined by the designer or operator. In our research, we use *relative standard error of the mean* (RSEM) to quantify the precision of a simulation result. RSEM represents the true mean of a population. For example, an RSEM of 5% for a mean value of 10 will provides simulation results with an accuracy of ± 0.5 units. The RSEM results are expressed in the same unit as of their data. Thus, smaller RSEM values provide more accurate results (better estimates) but require more simulation runs [110].

To calculate RSEM, first the standard error of the mean (SEM) is calculated using the following formula:

$$Standard\,error\,of\,mean\,(SEM) = \sigma/\sqrt{N}$$

The SEM takes into account the value of the standard deviation σ and the sample size N. Once the SEM is known, the RSEM is computed by dividing the SEM by the sample mean and multiply by 100 to express as a percentage.

$$Relative\,SEM = 100 * SEM/\bar{x}$$

In our current research, we use offline experiments to calculate the number of simulations required for a required RSEM. One of our future research goals is to automatically determine number of simulations required at runtime based for a given RSEM.

2.4 Systematic Empirical Inquiry

Empirical methods enable to provide evidence for the effectiveness and efficiency of proposed solutions. In our research, we use systematic empirical inquiry as an empirical method to evaluate our research. A systematic empirical inquiry allows to evaluate particular solutions using a rigorous approach that in a systematic manner define objectives, set up an experiment, collect and analyze data, and present the findings [81].

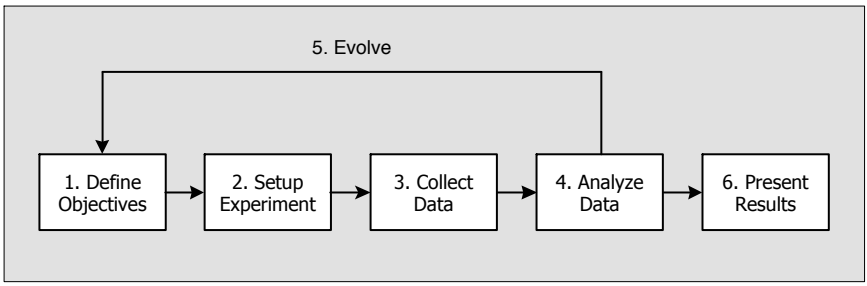


Figure 2.3: Overview of the systematic empirical inquiry used in our research

The stages of a systematic empirical inquiry are shown in Figure 2.3.

In stage 1, *define objectives*, we collect requirements of the domain for which self-adaptation is required. Basically, requirements are collected from stakeholders. In case of an exemplar, such as tele assistance and underwater unmanned vehicle, requirements are pre-defined. For other applications such as for Internet of Things (IoT) security monitoring application, requirements are collected from the stakeholders who have the domain knowledge. Once requirements are collected, these requirements are translated to adaptation goals to achieve. An example of an adaptation goal for the IoT application is the average packet loss over a period of 24 hours should not exceed 10%.

In stage 2, *setup experiment*, we realize the self-adaptation solution using ActivFORMS. In this stage, we prepare the system for self-adaptation that includes

creating probes and effectors (if not available), create formal models of the feedback loop that are deployed on top of the system. In the IoT application where system dynamics are slow, we created first a simulator that helped to perform experiments quicker before deploying the self-adaptation solution on the real system. For the tele assistance and IoT application we also realized a solution using runtime quantitative verification solution for comparison with ActivFORMS.

In stage 3, *collect data*, we run the experiments and collect the data. The data is collected using a simulator or on the real system (in case of IoT application). We also collect data for runtime quantitative verification when a comparison is required.

In stage 4, *analyze data*, we perform an analysis on the collected data. We use descriptive analysis techniques to summarize the results. In IoT application, we performed statistical analysis and determined p-values of the adaptation results using Wilcoxon and paired t-test to compare our results with reference and RQV approach.

In stage 5, *evolve*, we deal with evolution of the setting or requirements. In some cases, the evaluation is done incrementally, for example, a new goal is introduced by the stakeholders, or an additional experiment is required to test the scalability. In such a case, the first four steps are repeated, typically by adjusting the requirements and setting.

Finally, in stage 6, *present results*, we present the results usually in the form of a report or a scientific paper. All results are also made available via a website allowing other researcher to use our results or perform replications.

Chapter 3

ActivFORMS: Active Formal Models for Self-Adaptation

In this chapter, we introduce an initial version of the ActivFORMS approach; the chapter was published at 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, 2014. This chapter is a copy of [109].

The initial version of ActivFORMS addresses two research questions, i.e., RQ1 providing functional correctness of the feedback loop and RQ3 providing support for changing adaptation goals on the fly. Figure 3.1 shows an overview of the research goals addressed and the scientific methods used.

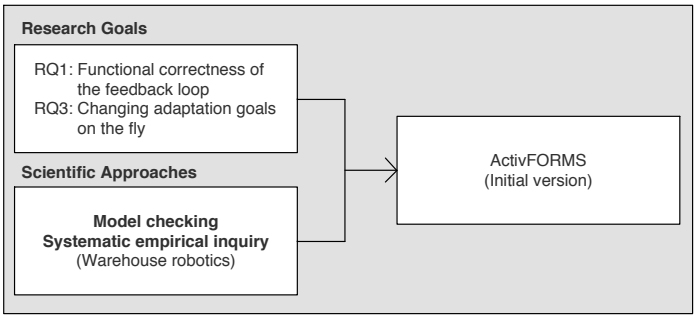


Figure 3.1: Overview of the addressed research goals and used scientific approaches in the initial version of ActivFORMS

To address RQ1, we formally specify and verify the correctness of the feedback loop model at design time using model checking. Then, the verified model of the feedback loop is directly executed at runtime using a virtual machine to realise adaptation. This way, the formal guarantees provided before deployment are preserved at runtime. The address RQ3 providing, we devised an approach that supports runtime updates of the adaptation goals and the feedback loop model. We integrated this approach into the virtual machine. The approach for live updates follows the classical process of quiescence [123], here applied to models.

My contribution to the research presented in this chapter is as follows: I contributed 70% of the conceptualisation, 100% of the technical realisation, and 50% of writing the chapter.

ActivFORMS: Active Formal Models for Self-Adaptation

Abstract

Self-adaptation enables a software system to deal autonomously with uncertainties, such as dynamic operating conditions that are difficult to predict or changing goals. A common approach to realize self-adaptation is with a MAPE-K feedback loop that consists of four adaptation components: Monitor, Analyze, Plan, and Execute. These components share Knowledge models of the managed system, its goals and environment. To provide guarantees of the adaptation goals, state of the art approaches propose using formal models of the knowledge. However, less attention is given to the formalization of the adaptation components themselves, which is important to provide guarantees of correctness of the adaptation behavior (e.g., does the execute component execute the plan correctly?). We propose Active FORmal Models for Self-adaptation (ActivFORMS) that uses an integrated formal model of the adaptation components and knowledge models. The formal model is directly executed by a virtual machine to realize adaptation, hence active model. The contributions of ActivFORMS are: (1) the approach assures that the adaptation goals that are verified offline are guaranteed at runtime, and (2) it supports dynamic adaptation of the active model to support changing goals. We show how we have applied ActivFORMS for a small-scale robotic system.

3.1 Introduction

Engineering the upcoming generation of software systems, such as networked smart homes and multi-robot systems, and guaranteeing the system goals during operation is complex due to uncertainties resulting from incomplete knowledge at design time. Among the uncertainties are changing availability of resources, dynamic operating conditions that are difficult to predict, and changing goals. Self-adaptation enables a software system to adapt autonomously to deal with such uncertainties. A self-adaptive system typically consists of a managed system and a feedback loop that adapts the managed system according to some goals. In this research, we focus on architecture-based self-adaptation [84, 122, 150] where adaptation is realized with a MAPE-K feedback loop that consists of four adaptation components: *Monitor*, *Analyze*, *Plan*, and *Execute*, complemented with *Knowledge* models of the managed system, its goals and environment [117, 211]. Monitor monitors the managed system and environment through probes, and updates Knowledge models accordingly. Analyze analyzes the data of the knowledge models and checks whether an adaptation is required. If so, it will trigger Plan that will compose a plan with actions that are then executed through effectors by Execute. Central to architecture-based adaptation is the separation of the domain concerns from the adaptation concerns; a recent controlled experiment [210] provides empirical evidence for the engineering benefits for this separation of concerns.

One important challenge in engineering self-adaptive systems is to provide evidence that the system goals are satisfied during operation, regarding the uncertainty of changes that may affect the managed system, its goals or environment [48, 134, 135]. To provide guarantees that the system goals are satisfied, state of the art in architecture-based self-adaptation advocates the use of formal models at runtime as one promising approach. In particular, existing approaches equip the feedback loop with formal models of the managed system and the environment in which it executes. A popular approach is using probabilistic models to model and handle uncertainties. The models are used to verify properties and support decision making about adaptation at runtime.

However, from a study of the state of the art we learned that existing approaches have payed little attention on providing guarantees about the behavior of the adaptation components themselves. For example, important properties of a self-healing system may be: does the analysis component correctly identify errors based on the monitored data, or does the execute component execute the actions to repair the managed system in the correct order? Lack of such guarantees may ruin the adaptation capabilities. In addition, we notice that little attention has been given on support for adaptation for changing goals or adding new goals at runtime. Existing approaches mainly focus on uncertainty with respect to parameters of knowledge models (failure rate of components, availability of a service etc.). Support for adaptation to deal with changing goals or adding new goals typically requires updates of the feedback loop.

In this paper, we propose Active FORmal Models for Self-adaptation (ActivFORMS). ActivFORMS contributes to the state of the art with an approach that guarantees the verified adaptation behavior at design time and provides first-class support for dealing with changing goals at runtime. The approach uses an integrated formal model of the complete MAPE-K loop, i.e., models of the knowledge and the adaptation components. The integrated formal model is directly executed by a virtual machine at runtime. We refer to this integrated formal model as *active model*. The active model can be dynamically changed with changing goals. The contributions of ActivFORMS are twofold. First, ActivFORMS assures that the goals that are verified offline are guaranteed at runtime. Second, the approach supports dynamic changes of the active model to support changing goals. This contrasts to existing approaches that provide guarantees during design, but require additional efforts to transfer the design into an actual implementation and maintain guarantees. In this paper, we focus on adaptation with MAPE-K feedback loops, but ActivFORMS can be applied to other types of feedback loops that can be modeled using the formal approach.

We evaluated ActivFORMS for a small scale system in which robots perform transportation tasks in a warehouse environment. The adaptation goal is to enable robots to adapt their behavior when a lane in the warehouse is temporally closed, for example for maintenance. We show that equipping each robot with an active model guarantees that the robots adapt their behavior correctly. The approach dynamically adapts the adaptation components to deal with potential deadlock due to a changing warehouse layout.

The remainder of this paper is structured as follows. Sect. 3.2 discusses the state of the art on the use of formal models for self-adaptation at runtime. In Sect. 3.3, we briefly introduce the robotic system. Sect. 3.4 presents the ActivFORMS approach and explains how we applied the approach to the robotic system. We reflect on the tradeoffs and restrictions of the approach in Sect. 3.5. Finally, we draw conclusions and outline possible future research in Section 3.6.

3.2 State of the Art

A recent systematic literature survey [207] covering the main software engineering venues between 2000 and 2012 identified a total of 75 papers that use formal methods in architecture-based self-adaptive systems. Among the primary studies, 25 study formal methods at runtime. We discuss a representative set of these papers and focus in addition on recent research results. We conclude with pointing out a number of interesting challenges in this area.

Back in 2002, Garlan and Schmerl [83] proposed an approach for model-based runtime adaptation of self-healing systems. Architecture models specified in Acme are checked via Armani, which evaluates first order constraints on the fly as properties of the architecture change. When problems are detected Armani triggers a repair engine to look for a repair strategy. This work laid the basis for the Rainbow framework [84].

[112] presents a process to create formal models for adaptive systems, verify the models and automatically translate the models into executable programs. The authors use Petri Nets and linear temporal logic to provide assurances for the system goals, and model-based testing to guarantee conformance between the models and programs. In follow up work [222], the authors model a dynamically adaptive program as a collection of (non-adaptive) steady-state programs and a set of adaptations that realize transitions among steady state programs in response to environmental changes. To handle the state explosion, the authors propose a modular model checking approach. Linear Temporal Logic (LTL) to specify properties of the non-adaptive portions of the system is combined with A-LTL (an adapt-operator extension to LTL) to concisely specify properties that hold during the adaptation process. This work provides an advanced approach for modular verification, however, its application at runtime needs further study.

[68] uses a probabilistic model (discrete time Markov chain) to represent an abstraction of the possible execution flows of a system at runtime. The probabilities that represent uncertainties are dynamically updated based on observations, using a Bayesian estimator. The probabilistic model can be used by a feedback loop to detect requirements violations and optimize the realization of system goals dynamically.

[72] proposes an approach for efficient runtime verification of reliability requirements. The proposed solution considers two distinct steps: pre-computation at design time and verification at runtime. The output of the pre-computation step is a set of symbolic expressions, which represent satisfaction of the requirements.

The verification step then evaluates the formula by replacing the variables with the runtime values gathered by monitoring the system.

[69] presents a quantitative approach for making adaptation decisions under uncertainty, called POISED. POISED builds on possibility theory (that is grounded in fuzzy mathematics) to assess both the positive and negative consequences of uncertainty. POISED makes adaptation decisions (runtime reconfiguration of its customizable software components) that result in the best range of potential behavior, improving a software system's quality of service.

[37] presents an advanced approach for self-adaptation to achieve QoS for service-based systems. Formally specified requirements are automatically analyzed to identify and enforce optimal system configurations by adapting service selection and resource allocation. The approach realizes a feedback loop based on MAPE-K, see Fig. 3.2. The Knowledge part of the feedback loop is modeled using

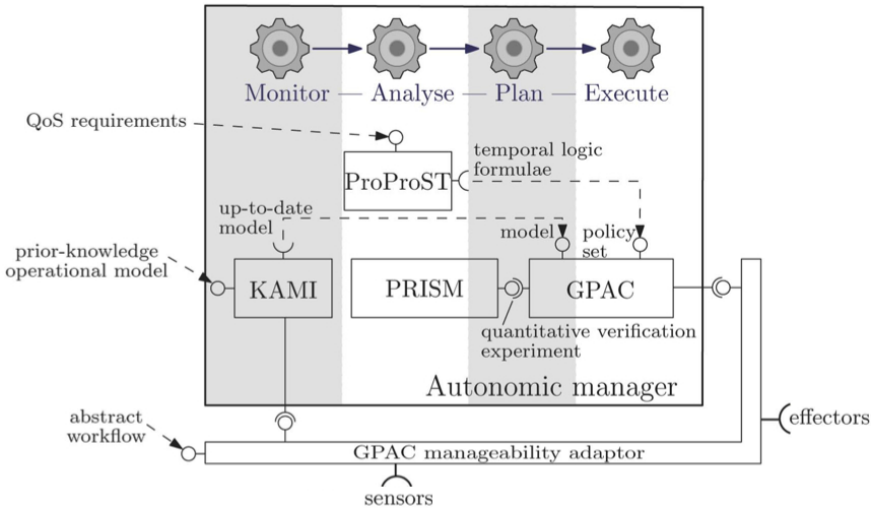


Figure 3.2: MAPE-K realization in [37]

different formal languages. The models are used by different tools to assure optimal reliability and performance requirements. As shown in the figure, the MAPE loop is realized by a series of tools that are glued together.

[91] introduces adaptive model-driven execution to mitigate uncertainties. In this approach, a Markov decision model of the system is generated from UML interaction diagrams. The model specifies the probability distribution of the different execution paths of the system. The model is executed by an interpreter that drives the execution of the system to guarantee the highest utility for a set of quality properties. The approach uses an embedded model to realize adaptation, but the concerns of the domain and adaptation are not clearly separated. The adaptation logic, which is encoded in the interpreter projects the possible future paths in the model to select the next action on the path with the highest utility.

Two interesting recently proposed approaches are [142] that focus on dynamic updates to deal with changing assumptions and requirements at runtime in time-

critical systems and [200] that proposes the EUREMA approach that realizes self-adaptation based on so called executable runtime mega-models.

Summary. To provide guarantees in self-adaptive systems, the use of formal models has gained increasing attention. Approaches that provide assurances by construction underpin the importance of formal methods. However, these approaches require additional efforts to provide guarantees of the actual implementation. Such efforts may be substantial to handle changing goals at runtime. For approaches that employ formal models at runtime, quantitative approaches are dominant and a number of studies support runtime verification. Virtually all studies focus on modeling the manged system, its environment, and the system goals (or parts of these). These models are then used by adaptation logic (i.e., the adaptation components) to reason about the system behavior and support analysis and decision making of adaptation actions. However, formal modeling of the adaptation components themselves and guaranteeing the required properties of the adaptation behavior has gained little attention. In a number of approaches, the realization of the adaptation logic includes tools, a prominent example is shown in Fig. 3.2. However, the behavior of the MAPE components and the integration of tools is often not formalized or verified. Furthermore, more research is required on adaptation to handle changing adaptation goals or adding new goals to the system at runtime, which represents an important class of uncertainty. Handling such changes typically requires dynamic updates of the adaptation components (and probably the managed system), which may require human involvement. Finally, model checking techniques are known to be computationally expensive, as they suffer from the state space explosion problem. Any solution based on such techniques, either used at design time or runtime is restricted with respect to providing guarantees in terms of system size. This also applies to ActivFORMS. To that end, research is required to realize scalable runtime verification for self-adaptive systems. However, this challenge is not the focus of the research presented in this paper.

3.3 Robotic System

Before we present ActivFORMS, we first briefly introduce the case study that we used in this research. The application consists of a set of robots that have to perform transportation tasks in a warehouse environment. Fig. 3.3 (left hand side) shows the user interface of the robotic system in simulation.

The map layout consists of a graph of connected nodes. In the example there are three source (or pick) locations (on the right hand side of the map marked with S) and two drop locations (on the left hand side marked with D). There are also two park locations (the zones at the top and bottom of the layout, see the figure). The tasks in this setting are performed by two robots. The robots receive tasks from a task managing system, or tasks can be manually added to the system. A task consists of picking a load at a pick location, drive to the drop location, and drop the load there. To avoid collisions and deadlock when they perform tasks, the

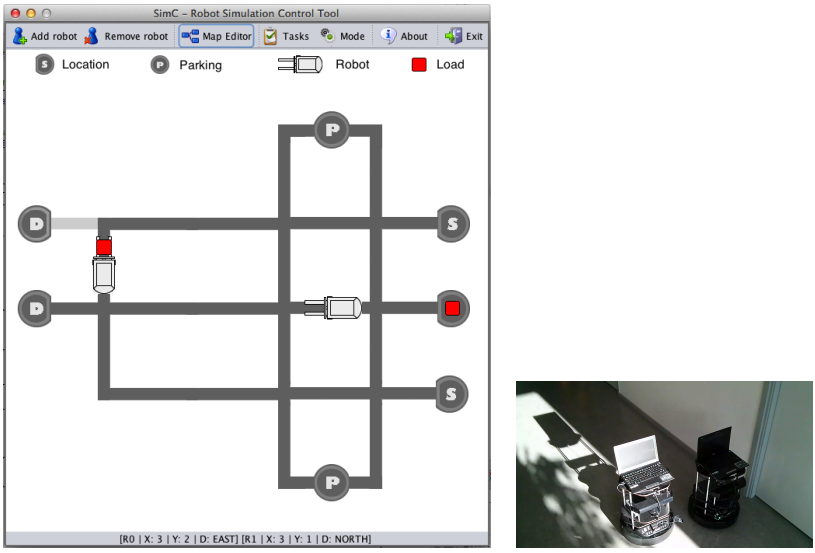


Figure 3.3: Left: User interface to the robot simulator. Right: Two Turtlebots in action.

robots can synchronize with one another to lock the next node they plan to visit. An idle robot can park at one of the park locations.

The robot system can operate in two modes: standard mode and shipping mode. In standard mode, the robots perform regular transportation tasks within the warehouse. In shipping mode, the robots have to perform tasks to load or unload a truck that can park at the drop location at the top. Mode changes are typically planned in advance, but an operator can also switch modes manually. In the depicted situation, the system is in standard mode (indicated by the light gray shade of the lane connected to the park location for trucks that is currently not accessible for the robots).

In this paper, we use self-adaptation to deal with lanes that have to be closed temporally in the warehouse, e.g., to perform maintenance tasks or to solve a problem with a robot. A lane can only be closed if none of the robots is depending on the lane for their current tasks. Closing a lane may also create the risk for deadlock, which needs to be anticipated.

To facilitate self-adaption capabilities, the robot program offers a monitor API that allows to retrieve the status of the robot (current position, current task, locked node, etc.) and an effector API that allows to perform adaptions of the robot (disable and enable a lane in the map of the robot, add and remove an element on the map, lock a node, etc.).

We are currently testing the scenarios with Turtlebot 2 robots (http://www.willowgarage.com/), see Fig. 3.3 (right hand side). The robot program to perform the transportation tasks is written in Java and is deployed on each robot. This program interacts with a local Python script that sends basic movement commands to the robot hardware using the Robot Operating System (ROS) API.

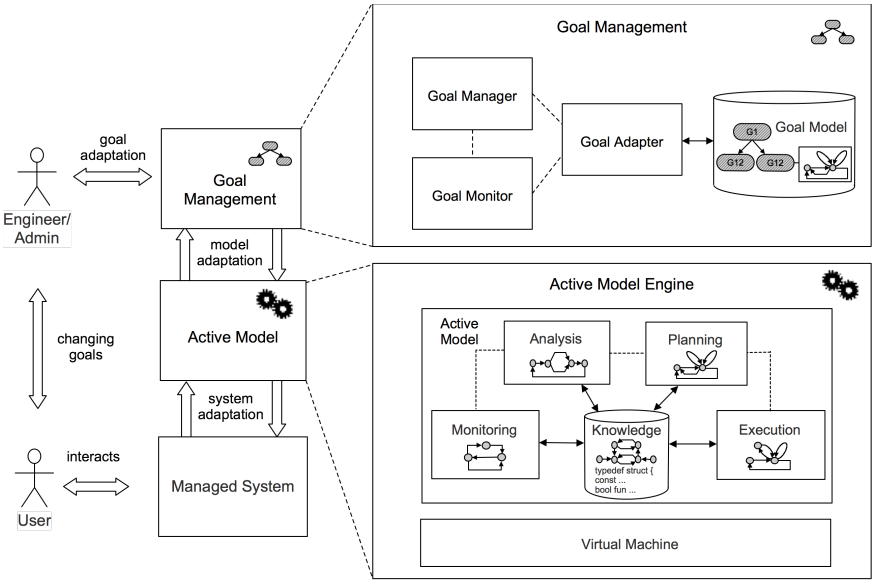


Figure 3.4: ActivFORMS approach

3.4 Approach

We now present the ActivFORMS approach. Fig. 3.4 shows the primary modules of ActivFORMS. The approach is in line with the three layered reference model for self-adaptive system proposed by Kramer and Magee [122]. The managed system realizes the domain functionality for users. In this research, we assume that the managed system is prepared to enable monitoring of relevant state and executing adaptation actions. Preparing the managed system for instrumentation to monitor and adapt the system is a research subject in its own right and out of scope of this paper. In the robotic system, the managed system is the robot program that enables the robot to perform transportation tasks. As explained in the previous section, the robot program provides monitoring and effector API's to facilitate extensions with self-adaptation capabilities. We now present the two central components of ActivFORMS: the active model engine and goal management.

3.4.1 Active Model Engine

The active model engine consists of two parts: an integrated formal model that realize a MAPE-K feedback loop, i.e., the active model, and a virtual machine that can execute the active model.

3.4.1.1 Active Model

In this research, we model feedback loops using networks of timed automata [19]. A timed automaton is a finite-state machine that models a behavior, extended with clock variables, which are used to synchronize behaviors. Automata can communicate through channels. There are two type of channels, binary channels and

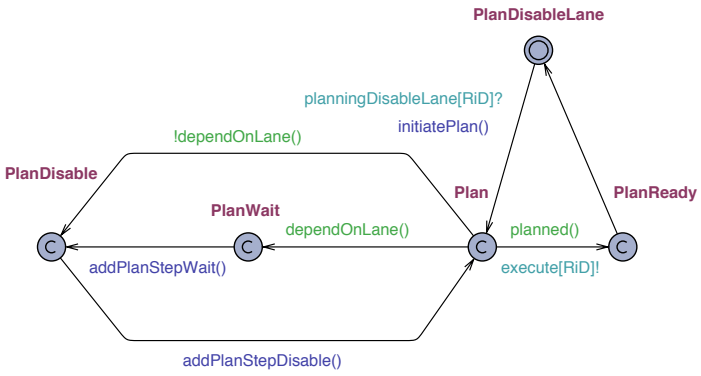


Figure 3.5: Plan automaton of a robot

broadcast channels. For a binary channel, a sender x! can synchronize with a receiver x? through a signal. If there are multiple receivers x? then a single receiver will be chosen non-deterministic. The sender x! will be blocked if there is no receiver. A broadcast channel sends a signal to all the receivers, and if there is no receiver, the sender will not be blocked. Behavior specifications can be complemented with expressions specified in a C-like language to define data structures (struct concept) and functions. Goals can be expressed in timed computation tree logic expressions (TCTL). TCTL expressions describe state and path formulae that can be verified, such as reachability (a system should/can/cannot/... reach a particular state or set of states), liveness (something eventually will hold), etc. We use Uppaal [16], a model checking tool that supports modeling of behaviors and verification of properties.

Fig. 3.4 shows an overview of the structure of the active model of a feedback loop design. The model consists of a network of timed automata (also called behaviors). The monitor behavior can receive data from probes that connect with the managed system. The execution behavior can send adaptation actions to effectors. The monitor, analyze, plan, and execute automata can interact directly via channels, or indirectly via reading and writing data in the knowledge. Knowledge can be represented with automata or with data structures (struct), or a combination of both. Fig. 3.5 shows a concrete example of an automaton of a robot that deals with the planning to disable a lane in the warehouse.

Planning starts when the automaton receives a signal from analysis (*planningDiableLane[RiD]?*). Planning first initiates a plan (*initiatePlan()*) and then checks whether the robot depends on the lane that has to be disabled, i.e, it may currently travel on the lane or it may need the lane to perform its current task (condition *dependOnLane()*). If that is the case, planning adds a step to the plan to let the robot wait until the condition no longer holds (*addPlanStepWait()*). Planning then moves on to *PlanDisable*. If the robot does not depend on the lane, planning immediately moves to *PlanDisable*. Planning then adds a step to the plan to disable the lane (*addPlanStepDisable()*), which completes planning. Finally, planning signals the execute behavior to execute the plan (*execute[Rid]!*).

Formal Guarantees. During design of the formal model, the required adaptation goals can be verified. To that end, the formal model of the MAPE-K feedback loop has to be connected with a models of the managed system. Evidently, the guarantees obtained from verification only hold to the extent that the implementation of the managed system conforms to the model of the system that is used for verification. Such conformance can be tested, for example with model-based testing techniques [189]. We illustrate verification with an example for the robotic system.

```
Monitoring(1).RequestToDisableLane
 && knowledge[1].disabledLane == Lane_cd
  --> Execution(1).DisableLane
  && knowledge[1].disabledLane == Lane_cd
```

The goal allows verifying that when the monitor behavior of robot with ID 1 receives a request for disabling a particular lane, the execution behavior will eventually adapt the managed system accordingly.

3.4.1.2 Virtual Machine

In ActivFORMS, the formally verified model can directly be executed to realize self-adaptation using a virtual machine. The virtual machine can perform the following functions: initiate model, execute model, interact with the managed system and the environment, verify goals at runtime, and update running models when requested. We discuss these functions in detail.

Initiate Model. When the virtual machine starts[1], it first translates the active model (network of automata) to an internal graph representation. Concretely, each node of an automaton becomes a node of a graph for that automaton; links between the nodes become edges between the corresponding nodes. Operations such as checking guards, updating state, etc. are translated into task graphs that are associated with the corresponding nodes and edges. Communication between automata (signals) are integrated in the task graphs of the edges or nodes that send and receive signals.

Fig. 3.6 shows an excerpt of the internal representation of the formal model of a robot. Fig. 3.6(a) shows the analysis automaton of a robot, (b) shows an excerpt of abstract syntax tree of the transition between the nodes *Analyzing* and *DisableLaneRequest*, and (c) shows the task graph generated for the guard *DiableLane == matchRequest(request)*. The task graph shows the subsequent atomic tasks that need to be executed to check the guard.

When the active model is translated to the internal representation, the state of each graph and the global state is initiated and the model is then ready for execution.

Execute the Model. Model execution conforms to the semantics of networked timed automata. The execution of the active model is triggered either by input from the managed system or the environment,[2] or by time.

[1] The virtual machine is implemented in Java and can be started with the ActivFORMSEngine class.

[2] The active model interacts with the managed system and the environment via signals that communicate with probes and effectors. We explain the details of interaction via probes and effectors below.

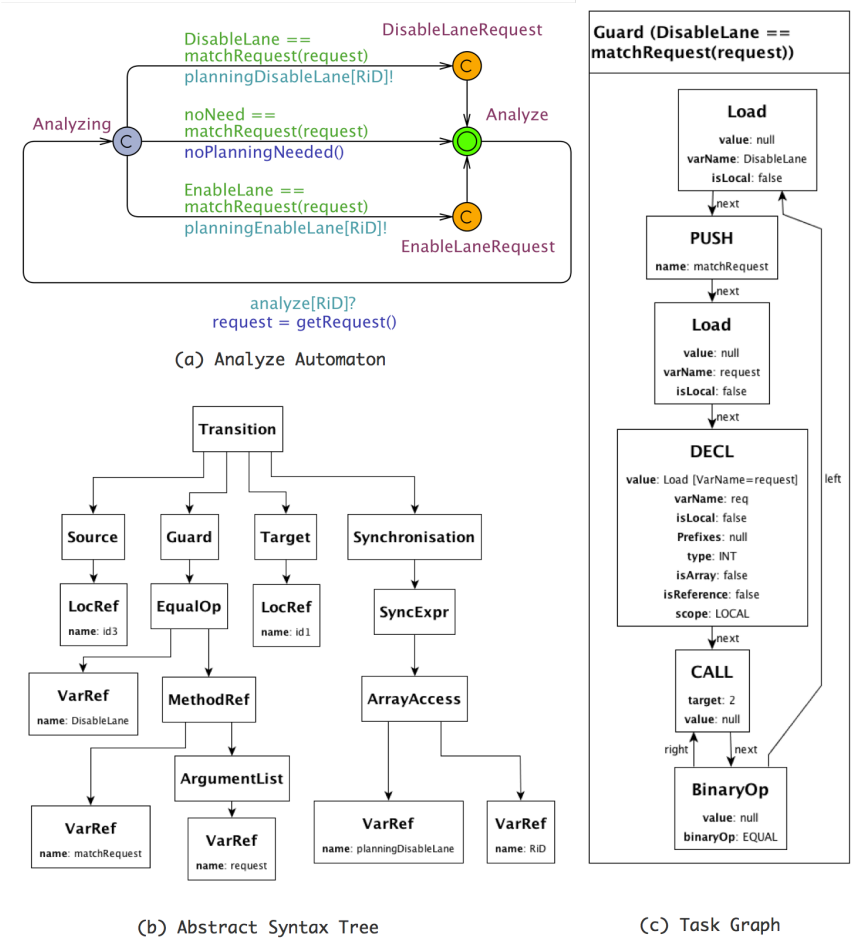


Figure 3.6: Excerpt of internal model representation

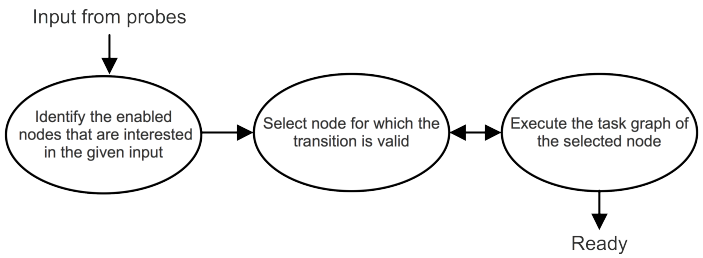


Figure 3.7: Input triggered execution

Fig. 3.7 shows the execution steps of an input triggered execution. When the virtual machine receives input, it first identifies the enabled nodes that are interested for the given input, which will be one or more nodes of the monitor behavior

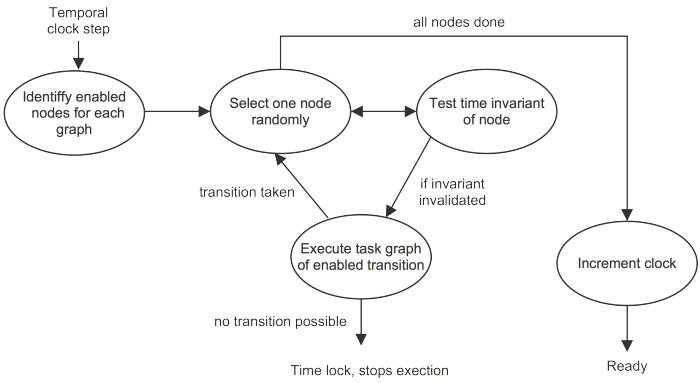


Figure 3.8: Time triggered execution

of the feedback loop. Next the virtual machine selects a node for execution.[3] If the transition is valid, the task graph of the selected node is executed, otherwise another enabled node is selected for execution. The execution of the task graph may trigger a subsequent behavior, e.g., a monitor may trigger an analysis and so forth. If there exists no valid node, no transition will be taken. This may point to an inconsistency between the model of the managed system that was used during the design of the managing system and the implementation of the managed system. An example of an input triggered execution in the robotic system is a monitor behavior that receives a signal from a probe to disable a lane and starts processing this request.

Fig. 3.8 shows the execution steps of a time triggered execution. The virtual machine maintains an internal clock that increments with time steps. The real time that corresponds with each time step can be configured in the virtual machine engine. In the following example, each step of the clock corresponds with 100 ms.

```
engine.setRealTimeUnit(100);
```

In line with the semantics of timed automata, for each time step, the virtual machine identifies the enabled node for each automaton and checks whether the time step would invalidate the time invariants of the enabled nodes. The virtual machine will then execute the task graphs of these invalidated nodes in non-deterministic order. If any of the nodes with a time invariant is not able to make a transition (due to a design flaw), time can no longer progress, which causes a timelock. In that case, a time lock exception will be thrown and execution of the model terminates. If no invalidated nodes exist, the clock will be increased and the execution step ends. An example of a time triggered execution in the robotic case is an execute behavior that executes the first step of the plan to disable a lane (the robot depends on the lane, see Fig. 3.5 and Fig. 3.12) and periodically checks whether the robot still depends on the lane.

[3] We limit the explanation to input triggered execution for binary channel semantics, that is, a signal that is sent from a probe synchronizes non-deterministic with one enabled location of a monitor automaton. The virtual machine also supports broadcast channels, where a signal can trigger multiple enabled locations.

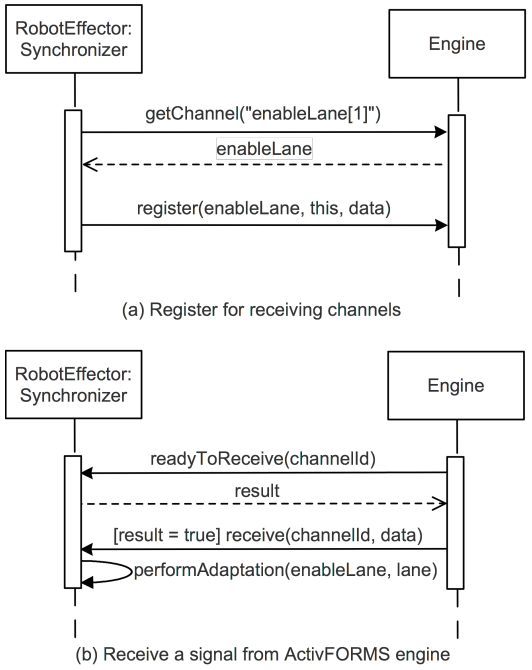


Figure 3.9: Interaction between an effector and the engine

Interaction with Managed System and Environment. The virtual machine interacts with the managed system and the environment through probes and effectors. For example, Fig. 3.9 shows two sequence diagrams that illustrate the interaction between the effector of a robot and the ActivFORMS engine to enable lanes.

To communicate with the virtual machine, the effector has to implement the *Synchronizer* interface and register to the virtual machine for channels of enabling lanes (Fig. 3.9(a)). When the execute behavior of the virtual machine wants to disable or enable a lane, it will synchronize with the effector using *readyToReceive()* (Fig. 3.9(b)). The execute behavior will then send the data of the adaptation action to the effector using *receive()*. Once the effector knows which lane' status has to be changed, it will execute the adaptation action using *performAdaptation()*. Probes work in a similar manner. The following excerpt shows how a probe communicates the updated position of a robot to the monitor.

```
engine.send(updatePosition, synch,
            "position.x=" + position.x,
            "position.y=" + position.y);
```

The parameter *updatePosition* represents the channel id, *synch* is a reference to return an acknowledgment when the *updatePosition* message is accepted. The last two parameters contain the data that the probe wants to send to the managing system, i.e., the current robot position. The virtual machine will process this message and execute the task graph of the enabled node of the monitor.

Verify Goals at Runtime. As formal verification is costly and exhaustive verification is hard to achieve, in particular when conditions are uncertain, verification at runtime may increase the evidence for assurances. Runtime verification can exploit the concrete information of the system and its environment that is available. ActivFORMS can exploit the direct availability of the formal model at runtime to support runtime verification. However, efficient runtime verification for self-adaptation remains a complex problem with many challenges. In the current version of ActivFORMS, we take a first step towards support for runtime verification. Concretely, the ActivFORMS engine currently supports the verification goals that can be specified as boolean expressions. To that end, the user has to specify the goal and load it into the virtual machine. The virtual machine will verify the goal in each execution step and notify the user whether the goal is satisfied or violated.

For example, a requirement for the robotic system is that a robot should never drive on a lane that is disabled:

```
notOn_disabledLane =
 "DISABLED_LANE == true &&
   knowledge[1].currentLane !=
      knowledge[1].disabledLane";
```

The virtual machine offers a method *addGoal()* to register goals:

```
engine.addGoal(notOn_disabledLane, client);
```

The *client* is the component that has an interest in the state of the goal.[4] When a goal is registered the virtual machine convert it into task graphs. After each transition, the virtual machine executes the task graphs of all registered goals and notifies the user whether the goals are satisfied or violated.

Changing Active Model at Runtime. There are two important motivations to provide support for changing the active model of the feedback loop at runtime. First, it enables efficient verification at runtime. Using specific models for the adaptation components that are tailored for different goals keeps the models small, which supports efficient verification. In the next section, we show how the goal manager automatically changes models of the feedback loop to deal with changing goals. A second important motivation is to support deployment of new models at runtime to deal with new goals. This latter typically involves humans in the loop. While this feature is supported by ActivFORMS, it is not the main focus of the research presented in this paper.

Fig. 3.10 shows the subsequent steps to change (parts of) a running active model. To start the change of an active model, the different parts of the model that need to be changed are loaded into the virtual machine using the method:

```
engine.changeModel(model);
```

The virtual machine then translates the model to the internal graph representation. The remaining steps to complete the dynamic change of the active

[4] In section 3.4.2, we will see how the Goal Manager of ActivFORMS serves as client to support changing goals.

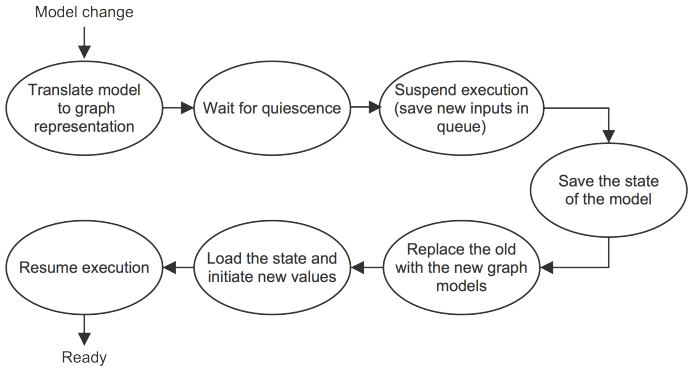


Figure 3.10: Change of the formal model at runtime

model essentially follow the classical process of runtime updates based on quiescence [123]. First, the virtual machine waits until the model reaches a quiescence state (i.e., no input or time triggered execution is ongoing). The virtual machine then suspends the execution. The state of the model is saved and new inputs are stored in a buffer. Subsequently, the old model is replaced by the new model. The saved state is restored and new variables are initiated. Finally, execution is resumed.

3.4.2 Goal Management

Goal management deals with adaption issues that cannot be handled by the current active model. Goal management consists of four key parts (see Fig. 3.4): goal model, goal monitor, goal adapter, and goal manager. We discuss each of them in detail.

Goal Model. The goal model represents the adaptation goals. We use tree-based models to specify goals. The goals at the bottom level of each subtree have associated models to realize adaptations. Fig. 3.11 shows an excerpt of a goal model for a robot.

The figure shows a goal tree to support adaptation for disabling lanes. The subtree on the left hand side combines *DISABLE_LANE == true* with two *TRANSPORT_MODE*, *standard* and *shipping* respectively. With each mode a corresponding formal model is associated. To support disabling of lanes in the standard mode, the virtual machine needs to execute the *Standard model*. However, in shipping mode, the *Shipping model* needs to be executed. To illustrate the need for different models, consider Fig. 3.12 that shows the models for the execute behavior for the two modes.

For standard mode, the execute behavior executes the steps of the plan to disable a lane (see the plan behavior of Fig. 3.5). However, in shipping mode, the map layout changes (a new lane and drop location is added, see Fig. 3.3), which creates new types of deadlock when a lane would be disabled.

Fig. 3.13 shows a schematic scenario for the robots in shipping mode. Assume the robots would have disabled the lane marked in dotted line. When robot R1

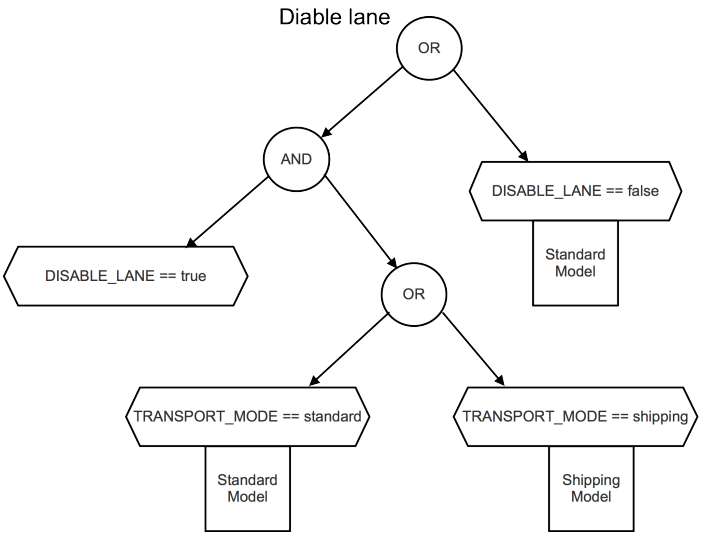


Figure 3.11: Goal model example

wants to drop a load at E and both robots have locked one node ahead, they end up in deadlock. To anticipate this problem, the new formal model of the feedback loop is required before the lane is disabled that avoids such deadlock situations. A concrete solution is to ensure that each robot locks two nodes before a lane can be closed. Therefore, the plan and execute behaviors have to be updated. Concretely, the planner in shipping mode has to add a extra step to the plan to lock an extra node. Fig. 3.12 shows the extra step that the execute behavior executes to lock an extra node, i.e., *executeLockNode()*. Locking an extra node constraints the mobility of the robots (it constraints path selection to drive), and should therefore only be applied when there is a request for disabling a lane in shipping mode.

Goal Monitor. The goal monitor monitors the status of the goals. To that end, the goal monitor adds the goals to the virtual machine and registers as client for notification of the status of the goals. The virtual machine keeps the goal monitor updated about the status of the goals. The goal monitor in turn will inform the goal adapter of any goal state changes and keep the goal manager up to date about the status of the goals (goal manager is discussed below).

Goal Adapter. The goal adapter is the heart of goal management. When the goal adapter is signaled by the goal monitor about a change of goals, it consults the goal model and search for a matching model that satisfies the changing situation. If the model associated with the changing goal differs from the currently deployed model, the goal adapter starts updating the current model with the new model at the virtual machine. If the model does not differ no further action is required. As an example, consider the Disable Lane goal shown in Fig. 3.11. If the robot system is running in standard mode and a lane is disabled, the standard model is running (left subtree of the goal). However, when the system switches to shipping

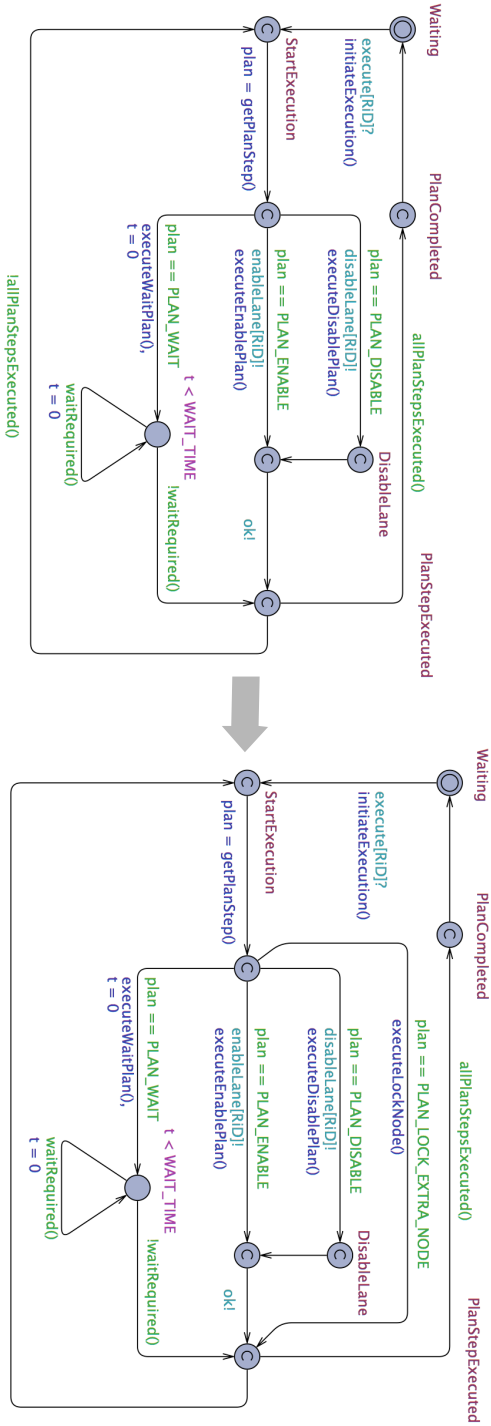


Figure 3.12: Execute behaviors for a robot in two operation modes

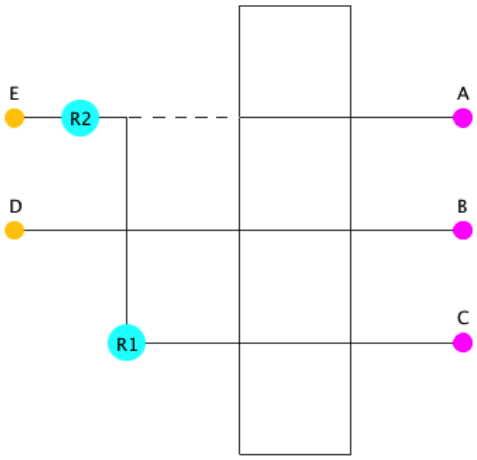


Figure 3.13: Deadlock scenario in robot system

mode, the running model has to be changed to shipping model. Once the new model is deployed, the system is ready to deal with deadlock when disabling lanes in shipping mode.

When the goal adapter finds no matching model that satisfies the changing goals, it will notify the goal manager. In this case, the adaptation goals cannot be satisfied and the goal manager will inform the system administrator.

Goal Manager. The goal manager offers support for three primary functions: inspecting the active model and its ongoing execution, monitoring and updating goals, and updating the goal model. In our current implementation, the ActivFORMS User Interface connects with the goal managers of the different nodes of the system. The user interface enables the system admin (or engineer) to perform the functions of the goal managers remotely. Fig. 3.14 shows the ActivFORMS User Interface.

The user interface allows a system admin to connect with goal managers of different nodes using the *Connect* button. In the snapshot, the user interface is connected with two robots. The main pane shows basic info about the robots and the status of their goals.

Clicking the *glasses* symbol for a robot shows the running model of the feedback loop of that robot, as illustrated in Fig. 3.15. The pane on the right hand side shows the current state of all data variables. The pane on the right hand side shows the models in action.

Fig. 3.16 shows the window that opens when the *Update Goals* button of the user interface is selected. The system admin can select a goal for any of the connected nodes in the left pane and edit the goal in the right pane. A new model can be selected and associated with the selected goal (*Save* button). Furthermore, new goals can be added or goals can be removed. All goal changes are directly forwarded to the corresponding goal manager to load the goal model. The model will be executed once it is selected by the goal adapter. To manually update the running

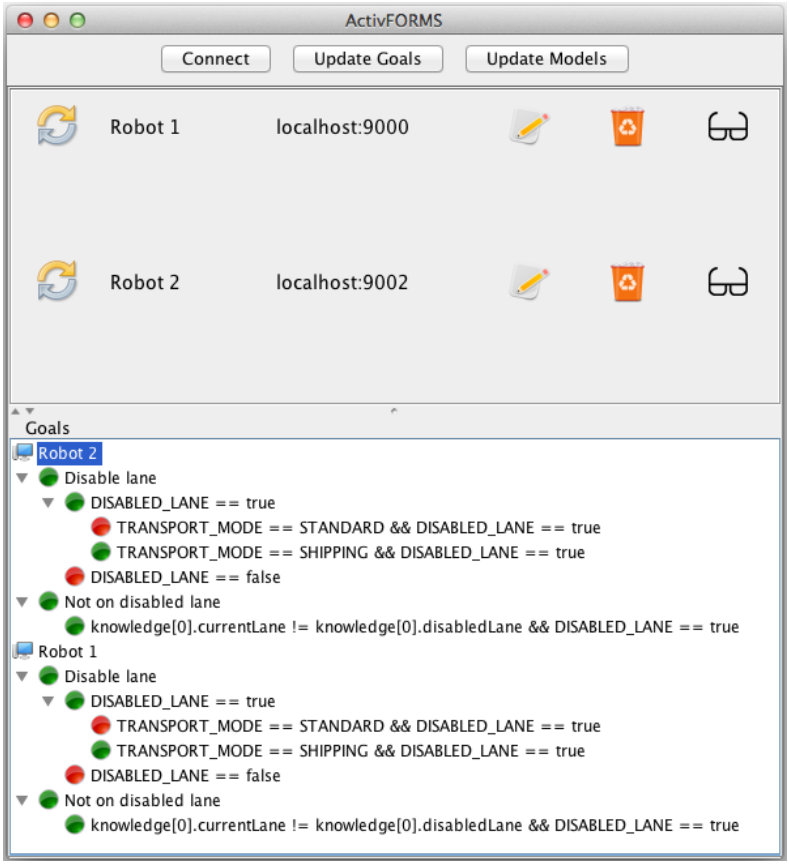


Figure 3.14: ActivFORMS User Interface

model, the admin can select the *Updates Models* button in the user interface. We do not further discuss this feature here.

3.5 Discussion

By directly executing the active model at runtime, ActivFORMS provides guarantees of the self-adaptive behavior and supports dynamic updates of the feedback loop. Active models go beyond the notion of model@runtime, which is defined [23] as a causally connected self-representation of the associated system [...] from a problem-space perspective. An active model is the *engine* that executes self-adaptation using a self-representation. Formalizing the distinct behaviors of a feedback loop supports fine grained verification of the correctness of the adaptation behaviors. Furthermore, modeling goals as first class entities and handling changing goals by dynamically changing the formal models of the adaptation behaviors supports small models that can be verified efficiently (at design time and potentially at runtime).

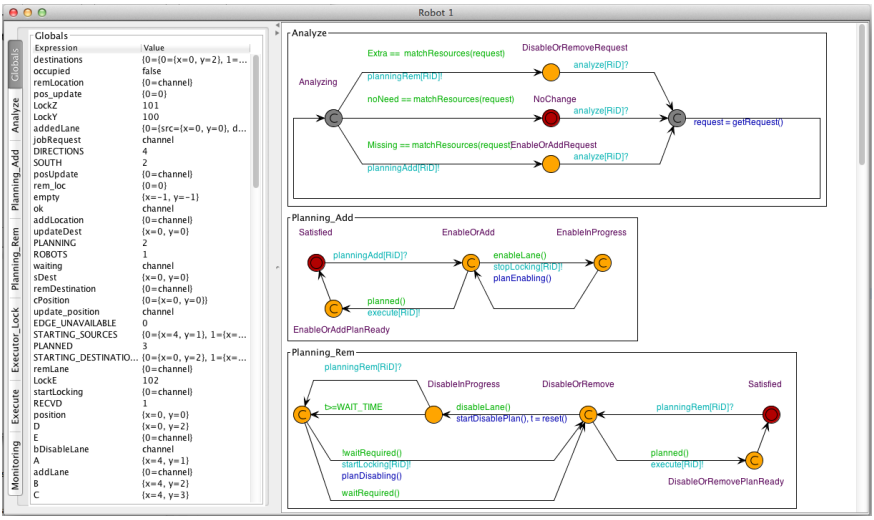


Figure 3.15: Illustration of an executing active model of a robot (red locations are current active locations)

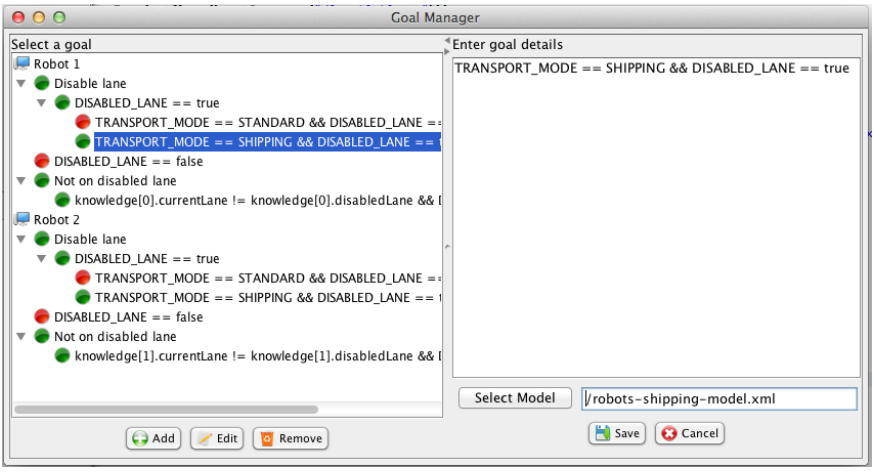


Figure 3.16: User interface for adding new goals

However, the proposed approach has a number of tradeoffs and restrictions. First, the approach requires expert knowledge to design and change the formal models, which is a characteristic of every approach where designers have to use formal methods. ActivFORMS uses timed automata and TCTL, which offer an accessible notation. To further support designers with modeling MAPE-K feedback loops, we have developed a set of templates for designing the behaviors of the MAPE-K components.[1] These templates define abstract automata that can be

[1] Available at the ActivFORMS website:
https://people.cs.kuleuven.be/danny.weyns/software/ActivFORMS/

instantiated and extended for the domain at hand. The templates were derived from experience with modeling MAPE-K feedback loops for different applications, e.g., [94, 108]. Nevertheless, formal modeling remains a tedious task. In our future work, we plan to explore how the underlying formalism could be hided from designers. An inspiring approach is proposed in [91]. Second, in this paper, we have used ActivFORMS to realize MAPE-K loops, but ActivFORMS can execute other types of feedback loops [28] as long as the components of the loops can be modeled correctly with timed automata. Third, the approach relies on models. However, accurate information to the design the models may not be available at design time. To support modeling of uncertainty of the environment and the system in the knowledge part of the feedback loop, we are currently working on extending ActivFORMS with probabilistic timed automata. Fourth, timed automata may not be an appropriate language for modeling the behavior of the feedback loop for particular types of systems; an example is a system that requires continuous adaptation features. Fifth, ActivFORMS approach is not tested yet for medium or large-scale systems. One feature of the approach to handle scalability is dynamic switching of models to handle different goals, which keeps the running models small. Nevertheless, further study is required to study the scalability of the approach. Sixth, ActivFORMS introduces some overhead. The memory required to launch the virtual machine engine and load a active model is approximately 100 MB, depending on the size of the active model that is used. Performance overhead may be an issue, as the virtual machine has to check the validity of the transitions of the enabled nodes in each execution step. For the robotic system, the performance overhead was minimal, but this might be different for other domains.

3.6 Conclusions & Future Work

In this paper, we presented ActivFORMS, a formal approach for self-adaptation. ActivFORMS distinguishes itself for existing approaches in two ways. First, the formally verified model of the complete feedback loop is directly executed, which guarantees the verified adaptation goals at runtime. This contrasts in particular to existing approaches that provide guarantees during design, but require additional efforts to transfer the design into an actual implementation and assure guarantees. As the active model is directly executed in ActivFORMS, the approach does not require coding. We recently performed a series of case studies in the context of a Master course on Adaptive Systems. Initial results show that the total time for adding a self-adaptation property to a legacy system was up to 3 times lower when using ActivFORMS comparing to regular coding of the system.

ActivFORMS considers goals as first-class citizens and is designed to support runtime updates of the feedback loop, which allows to dynamically change models of the adaptation components handling changing goals. Deploying small models enables focussed verification, which is important to deal with the state space problem inherent to verification. Furthermore, supporting dynamic updates is important to deal with uncertainty, in particular dynamically adding new goals. It is

noteworthy to mention that runtime updates of the active model is a lightweight process.

ActivFORMS paves the way for several lines of future research. In our current work, we are extending the virtual machine to support probabilistic timed automata. The current version of ActivFORMS supports online detection of simple goal violations. We will enhance the support for more advanced goal models and study how to handle dependencies between goals. In the future, we also plan to study how we can introduce efficient runtime verification in ActivFORMS. Our aim is to enhance ActivFORMS with a plugin for incremental verification at runtime, which would allow verifying goals within a restricted time window. Another line of research that we plan to explore is supporting coordination between formal models. Currently, an active model interacts only with the local managed system and its environment. Supporting interactions between formal models would open both a way to handle multiple concerns locally and coordination between distributed active models in a decentralized setting.

Chapter 4

Engineering Trustworthy Self-Adaptive Software with Dynamic Assurance Cases

In this chapter, we introduce ENgineer TRUstworthy Self-adaptive sofTware systems (ENTRUST), an integrated methodology to engineer self-adaptive systems with strict requirements. This chapter is accepted for publication in IEEE Transactions on Software Engineering, 2017. This chapter was a joint effort between three research teams (R. Calinescu and S. Gerasimou, D. Weyns and M. U. Iftikhar, T. Kelly and I. Habli). This chapter is a copy of [40].

This chapter demonstrates how ActivFORMS can be incorporated with other approaches. Figure 4.1 provides an overview of contribution of ActivFORMS to the ENTRUST approach.

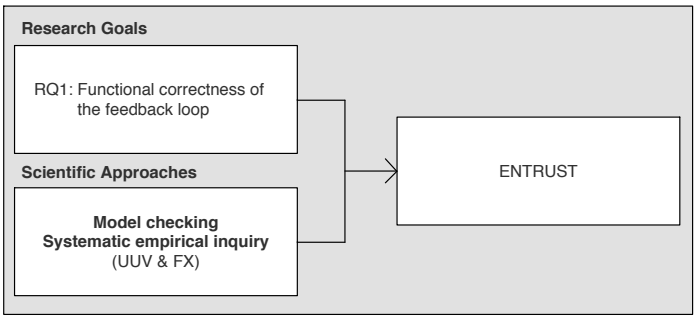


Figure 4.1: Overview of contribution of ActivFORMS to ENTRUST approach

The primary contribution of ActivFORMS to ENTRUST is the realisation of functional correctness of the feedback loop (RQ1). To that end, we contributed an initial set of formally specified templates that help engineers to design and verify MAPE feedback loop models (based on [79]). We evaluated ENTRUST with an Unmanned Underwater Vehicle (UUV) System and Foreign Exchange Trading (FX) System.

My contribution to the research presented in this chapter is as follows: I contributed 35% of the overall technical realisation of ENTRUST, 35% of the evaluation, and 10% of writing the chapter.

Engineering Trustworthy Self-Adaptive Software with Dynamic Assurance Cases

Abstract

Building on concepts drawn from control theory, *self-adaptive software* handles environmental and internal uncertainties by dynamically adjusting its architecture and parameters in response to events such as workload changes and component failures. Self-adaptive software is increasingly expected to meet strict functional and non-functional requirements in applications from areas as diverse as manufacturing, healthcare and finance. To address this need, we introduce a methodology for the systematic ENgineering of TRUstworthy Self-adaptive sofTware (ENTRUST). ENTRUST uses a combination of (1) design-time and runtime modelling and verification, and (2) industry-adopted assurance processes to develop trustworthy self-adaptive software *and* assurance cases arguing the suitability of the software for its intended application. To evaluate the effectiveness of our methodology, we present a tool-supported instance of ENTRUST and its use to develop proof-of-concept self-adaptive software for embedded and service-based systems from the oceanic monitoring and e-finance domains, respectively. The experimental results show that ENTRUST can be used to engineer self-adaptive software systems in different application domains and to generate dynamic assurance cases for these systems.

4.1 Introduction

Software systems are regularly used in applications characterised by uncertain environments, evolving requirements and unexpected failures. The correct operation of these applications depends on the ability of software to adapt to change, through the dynamic reconfiguration of its parameters or architecture. When events such as variations in workload, changes in the required throughput or component failures are observed, alternative adaptation options are analysed, and a suitable new software configuration may be selected and applied.

As software adaptation is often too complex or too costly to be performed by human operators, its automation has been the subject of intense research. Using concepts borrowed from the control of discrete-event systems [160], this research proposes the extension of software systems with *closed-loop control*. As shown in Fig. 4.2, the paradigm involves using an external software *controller* to monitor the system and to adapt its architecture or configuration after environmental and internal changes. Inspired by the autonomic computing manifesto [105, 117] and by pioneering work on self-adaptive software [114, 151], this research has been very successful [203]. Over the past decade, numerous research projects proposed architectures [84, 122, 215] and frameworks [37, 67, 188, 211] for the engineering of *self-adaptive systems*. Extensive surveys of this research and its applications are available in [106, 157, 164].

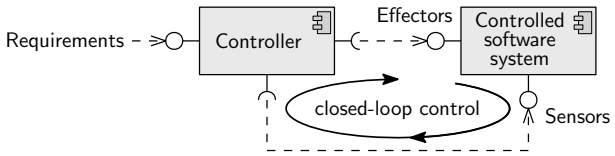


Figure 4.2: Closed-loop control is used to automate software adaptation

In this paper, we are concerned with the use of self-adaptive software in systems with strict functional and non-functional requirements. A growing number of systems are expected to fit this description in the near future. Service-based telehealth systems are envisaged to use self-adaptation to cope with service failures and workload variations [37, 68, 204], avoiding harm to patients. Autonomous robots used in applications ranging from manufacturing [63, 90] to oceanic monitoring [34, 87] will need to rely on self-adaptive software for completing their missions safely and effectively, without damage to, or loss of, expensive equipment. Employing self-adaptive software in these applications is very challenging, as it requires assurances about the correct operation of the software in scenarios affected by uncertainty.

Assurance has become a major concern for self-adaptive software only recently [7, 49, 134, 135, 166]. Accordingly, the research in the area is limited, and often confined to providing evidence that individual aspects of the self-adaptive software are correct (e.g. the software platform used to execute the controller, the controller functions, or the runtime adaptation decisions). However, such evidence is only one component of the established industry process for the assurance of software-based systems [24, 137, 192]. In real-world applications, assuring a software system requires the provision of an *assurance case*, which standards such as [193] define as

> "*a structured argument, supported by a body of evidence, that provides a compelling, comprehensible and valid case that a system is safe for a given application in a given environment*".

Our work addresses this discrepancy between the state of practice and the current research on assurances for self-adaptive software. To this end, we introduce a generic methodology for the joint development of trustworthy self-adaptive software systems *and* their associated assurance cases. Our methodology for the ENgineering of TRUstworthy Self-adaptive sofTware (ENTRUST) is underpinned by a combination of (1) design-time and runtime modelling and verification, and (2) an industry-adopted standard for the formalisation of assurance arguments [95, 178].

ENTRUST uses design-time modelling, verification and synthesis of assurance evidence for the control aspects of a self-adaptive system that are engineered before the system is deployed. These design-time activities support the initial controller enactment and the generation of a partial assurance case for the self-adaptive system. The dynamic selection of a system configuration (i.e., architecture and parameters) during the initial deployment and after internal and environmental

changes involves further modelling and verification, and the synthesis of the additional assurance evidence required to complete the assurance case. These activities are fully automated and carried out at runtime.

The ENTRUST methodology is not prescriptive about the modelling, verification and assurance evidence generation methods used in its design-time and runtime stages. This generality exploits the fact that the body of evidence underpinning an assurance case can combine verification evidence from activities including formal verification, testing and simulation. As such, ENTRUST assurance cases can use assurance evidence obtained through a combination of testing, simulation and formal verification, at both design time and runtime.

ENTRUST supports the systematic engineering and assurance of self-adaptive systems. In line with other research on self-adaptive systems (see e.g. [164, 207]), we assume that the controlled software system from Fig. 4.2 already exists, and we focus on its enhancement with self-adaptation capabilities through the addition of a high-level monitor-analyse-plan-execute (MAPE) control loop. The components of the controlled software system may already support low-level, real-time adaptation to localised changes. For instance, the self-adaptive embedded system used in one of our case studies is a controlled unmanned vehicle that employs built-in low-level control to maintain the speed selected by its high-level ENTRUST controller. Mature approaches from the areas of robust control of discrete-event systems (e.g. [127, 160, 183, 220]) and real-time systems (e.g. [124, 138]) already exist for the engineering of such low-level control. Thus, real-time control is outside the scope of ENTRUST.

Likewise, established assurance processes are available for the non-self-adaptive aspects of software systems (e.g. [22, 24, 98, 100, 163]). We do not duplicate this work. Using these processes to construct assurance arguments for the correct design, development and operation of the controlled software system, and for the derivation, validity, completeness and formalisation of the requirements from Fig. 4.2 is outside the scope of our paper. Thus, ENTRUST focuses on the correct engineering of the controller and on the correct operation of self-adaptive system, assuming that the controlled system and its requirements are both correct.

The main contributions of our paper are:

1) The first end-to-end methodology for (a) engineering self-adaptive software systems with assurance evidence for the controller platform, its functions and the adaptation decisions; and (b) devising assurance cases whose assurance arguments bring together this evidence.
2) A novel assurance argument pattern for self-adaptive systems, expressed in the Goal Structuring Notation (GSN) standard [95] that is widely used for assurance case development in industry [178].
3) An instantiation of our methodology whose stages are supported by the established modelling and verification tools UPPAAL [17] and PRISM [126].

These contributions include four significant extensions of complementary results from our previously separate strands of work on developing formally verified control loops [79, 109], runtime probabilistic model checking [38] and dynamic assurance cases [61]. First, the instantiation of the ENTRUST methodology is based

on a formally verifiable controller architecture where the controller from [109] was extended to use probabilistic model checking at runtime [38]. Second, building on [79], we introduce a set of generic properties that ENTRUST controllers must satisfy. Third, we extend our preliminary work from [61] with a realisation of the principles of dynamic assurance case continuity, updatability, proactivity, automation and formality that we suggested in [61]. Fourth, we devise the first assurance argument pattern for self-adaptive systems. In addition, we integrate these extended building blocks into a complete methodology for the engineering of self-adaptive systems.

To ensure the generality of ENTRUST, these contributions are evaluated using two case studies with different characteristics (e.g. types of system, requirements and adaptation actions) and belonging to different application domains (i.e. oceanic monitoring and exchange trade). We chose for these case studies systems that have been used to evaluate related software engineering research [34, 87, 88, 171], as these systems are already known to the research community – one of them as an "exemplar" for the evaluation of new approaches to engineering self-adaptive systems [89].

The remainder of the paper is organised as follows. In Section 4.2, we provide background information on assurance cases, GSN and assurance argument patterns. Section 4.3 introduces the self-adaptive systems used in our case studies, and Section 4.4 describes the generic ENTRUST methodology. Sections 4.5 and 4.6 present the tool-supported ENTRUST instance and its use to develop the self-adaptive systems from the two case studies, respectively. Section 4.7 presents our evaluation results, which show that the methodology can be used for the effective engineering of self-adaptive systems from different domains and for the generation of dynamic assurance cases for these systems. In Section 4.8, we overview the existing approaches to providing assurances for self-adaptive software systems, and we compare them to ENTRUST. Finally, Section 4.9 concludes the paper with a discussion and a summary of future work directions.

4.2 Preliminaries

This section provides background information on assurance cases, introducing the assurance-related terminology and concepts used in the rest of the paper. We start by defining assurance cases and their components in Section 4.2.1. Next, we introduce a commonly used notation for the specification of assurance cases in Section 4.2.2. Finally, we introduce the concept of an assurance argument pattern in Section 4.2.3.

4.2.1 Assurance Cases

An *assurance case*[1] is a report that supports a specific *claim* about the requirements of a system [22]. As an example, the assurance case in [149] provides documented

[1] Assurance cases developed for safety-critical systems are also called *safety cases*. In this work, we are concerned with any self-adaptive software systems that must meet strict requirements, and therefore we talk about assurance cases and assurance arguments.

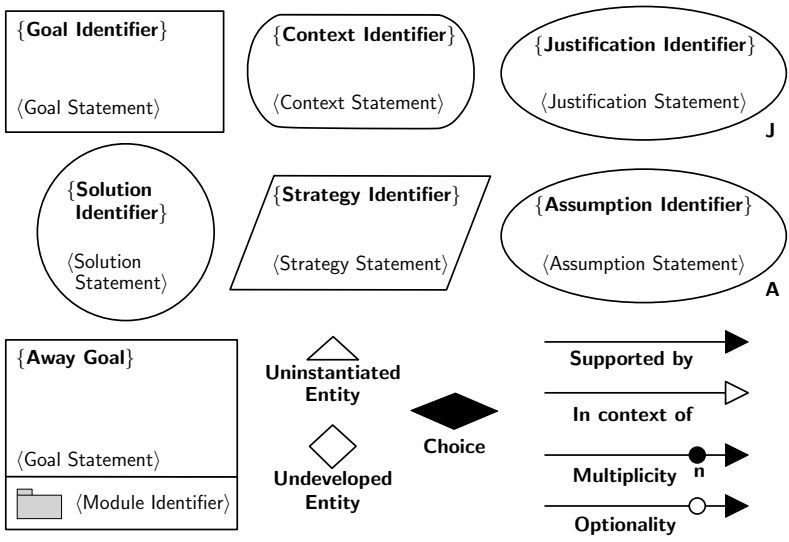


Figure 4.3: Core GSN elements

assurance that the "*implementation and operation of North European Functional Airspace Block (NEFAB) is acceptably safe according to ICAO, EC and EUROCONTROL safety requirements.*" The documented assurance within an assurance case comprises (1) *evidence* and (2) structured *arguments* that link the evidence to the claim [22], possibly through intermediate claims.

Assurance cases are becoming mandatory for software systems used in safety-critical and mission-critical applications [24, 137, 192]. They are used in domains ranging from nuclear energy [194] and medical devices [196] to air traffic control [70] and defence [193]. A growing number of assurance cases from these and other domains are openly available (e.g., [149, 195]).

The development of assurance cases comprises processes carried out at all stages of the system life cycle [192]. Requirements analysis evidence and design evidence demonstrate that system reliability, safety, maintainability, etc. are considered in the early stages of the life cycle. Implementation, validation and verification evidence are then generated as the system is developed. Finally, evidence collected at runtime is used to update assurance cases during system maintenance.

As aptly described in [192], the assurance case must be "*a living, cradle-to-grave document.*" This is particularly true for self-adaptive software systems. For these systems, existing evidence needs to be continuously combined with new *adaptation evidence*, i.e., evidence that the system will continue to operate safely after self-adaptation activities.

4.2.2 Goal Structuring Notation

The assurance cases for self-adaptive systems introduced later in the paper are devised in the *Goal Structuring Notation* (GSN) [115], a community standard [95] widely used for assurance case development in industry [178]. The main GSN

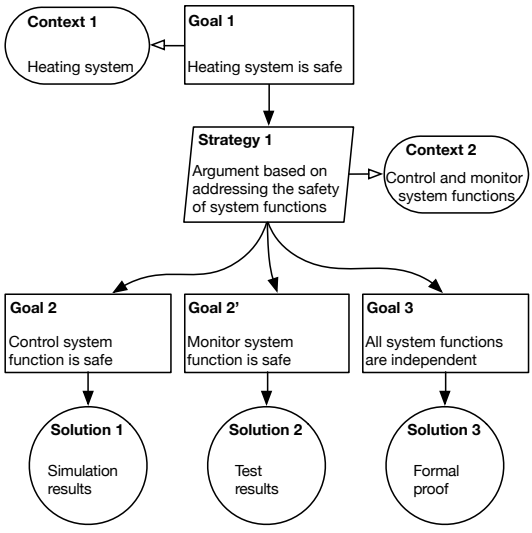


Figure 4.4: Example of a GSN assurance argument

elements (Fig. 4.3) can be used to construct an argument by showing how an assurance claim (represented in GSN by a *goal*) is broken down into sub-claims (also represented by GSN *goals*), until eventually it can be supported by GSN *solutions* (i.e., assurance evidence from verification, testing, etc.). *Strategies* are used to partition the argument and describe the nature of the inference that exists between a goal and its supporting goal(s). The rationale (*assumptions* and *justifications*) for individual elements of the argument can be captured, along with the *context* (e.g. to describe the operational environment) in which the claims are stated.

In a GSN diagram, claims are linked to strategies, sub-claims and ultimately to solutions using '*supported by*' connectives, which are rendered as lines with a solid arrowhead and declare inferential or evidential relationships. 'Supported by' connectives may be decorated with their multiplicity or marked as optional. The '*in context of*' connective, rendered as a line with a hollow arrowhead, declares a contextual relationship between a goal or strategy on the one hand and a context, assumption or justification on the other hand.

Large or complex sections of the assurance argument can be organised into modules by means of GSN *away goals* referenced in the main argument and defined separately. Finally, GSN entities can be marked as *uninstantiated* to indicate that they are placeholders that need to be replaced with a concrete instantiation, and GSN goals can be marked as *undeveloped* to indicate that they need to be further developed into sub-goals, strategies and solutions.

As an example, Fig. 4.4 shows a simple GSN assurance argument for the software part of a heating system. Its root goal (**Goal 1**) claims that the system is safe at all times. This claim is partitioned into sub-claims using a strategy (**Strategy 1**) that addresses the safety of the two system functions (i.e. control and monitoring) separately through sub-claims **Goal 2** (for the control system) and **Goal 2'** (for

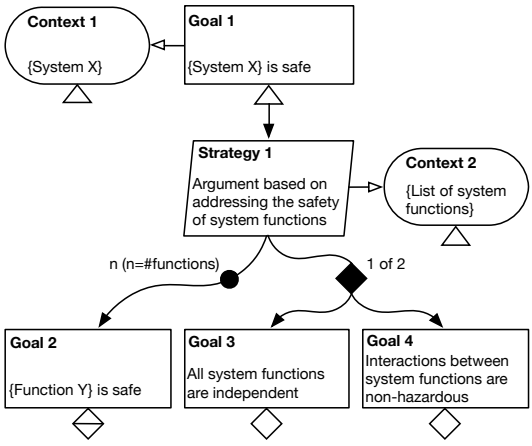


Figure 4.5: Example of a GSN assurance argument pattern

the monitor system), and includes sub-claim **Goal 3** that the two functions are independent. The three sub-claims are supported by three solutions comprising assurance evidence from simulation, testing and formal proof, respectively.

4.2.3 Assurance Argument Patterns

To reduce the significant effort required to develop assurance cases, in our previous work on software assurance [98, 99] we collaborated to the creation of a catalog of reusable GSN *assurance argument patterns* [101]. Each pattern considers the contribution made by the software to system hazards for a particular class of systems and scenarios. The GSN elements of a pattern that are generic to the entire class are fully developed and instantiated, whereas the entities that are specific to each system and scenario within the class are left undeveloped and/or uninstantiated.

As an example, Fig. 4.5 depicts an assurance argument pattern that is instantiated by the GSN assurance argument from Fig. 4.4. The elements surrounded by curly brackets '{' and '}' in the pattern must be *instantiated* for each assurance argument based on the pattern, as further indicated by the triangular 'uninstantiated' symbol under the GSN entities that contain them. **Goal 2** is marked with both this 'uninstantiated' symbol (because it contains elements in curly brackets) and a diamond-shaped 'undeveloped' symbol (because, like for the 'choice' sub-claims **Goal 3** and **Goal 4**, additional GSN entities must be added underneath to complete the assurance argument); the two symbols are rendered overlapping under **Goal 2**.

In this paper, we devise a new assurance argument pattern, which is applicable to self-adaptive software systems.

4.3 Self-adaptive Systems Used in the Case Studies

This section introduces the self-adaptive software systems from the two case studies used to illustrate and evaluate our methodology. To assess the generality of

Table 4.1: Comparison of systems used to assess the generality of ENTRUST

	UUV	**FX**
Type	embedded system	service-based system
Domain	oceanic monitoring	exchange trade
Requirements	throughput, resource use, cost, safety	reliability, response time, cost, safety
Sensor data	UUV sensor measurement rate	service response time and reliability
Adaptation actions	switch sensors on/off, change speed	change service instance
Uncertainty modelling	continuous-time stochastic model of UUV sensors	discrete-time stochastic model of system

ENTRUST, we chose different types of systems from different domains. The first system, introduced in Section 4.3.1, is an embedded unmanned underwater vehicle (UUV) system from the oceanic monitoring domain. The second system, presented in Section 4.3.2, is a service-based system from the foreign exchange (FX) trade domain. Table 4.1 lists several additional characteristics that differ significantly between the two systems. These characteristics include the types of requirements, sensor data and adaptation actions of the systems, and the types of models whose verification underpins their self-adaptation decisions.

4.3.1 Unmanned Underwater Vehicle (UUV) System

The self-adaptive UUV embedded system is adapted from [87]. UUVs are increasingly used in a wide range of oceanographic and military tasks, including oceanic surveillance (e.g., to monitor pollution levels and ecosystems), undersea mapping and mine detection. Limitations due to their operating environment (e.g., impossibility to maintain UUV-operator communication during missions and unexpected changes) require that UUV systems are self-adaptive. These systems are often mission critical (e.g., when used for mine detection) or business critical (e.g., they carry expensive equipment that should not be lost).

The self-adaptive system we use consists of a UUV deployed to carry out a data gathering mission. The UUV is equipped with $n \geq 1$ on-board sensors that can measure the same characteristic of the ocean environment (e.g., water current, salinity or temperature). When used, the sensors take measurements with different, variable rates $r_1, r_2, \ldots, r_n$. The probability that each sensor produces measurements that are sufficiently accurate for the purpose of the mission depends on the UUV speed sp, and is given by $p_1, p_2, \ldots, p_n$. For each measurement taken, a different amount of energy is consumed, given by $e_1, e_2, \ldots, e_n$. Finally, the n sensors can be switched on and off individually (e.g., to save battery power when not required), but these operations consume an amount of energy given by $e_1^{\text{on}}, e_2^{\text{on}}, \ldots, e_n^{\text{on}}$ and $e_1^{\text{off}}, e_2^{\text{off}}, \ldots, e_n^{\text{off}}$, respectively. The UUV must adapt to changes in the sensor measurement rates $r_1, r_2, \ldots, r_n$ and to sensor failures by dynamically adjusting:

(a) the UUV speed sp

(b) the sensor configuration $x_1, x_2, \ldots, x_n$ (where $x_i = 1$ if the i-th sensor is on and $x_i = 0$ otherwise)

in order to meet the quality-of-service requirements below:

R1 (throughput): The UUV should take at least 20 measurements of sufficient accuracy for every 10 metres of mission distance.

R2 (resource usage): The energy consumption of the sensors should not exceed 120 Joules per 10 surveyed metres.

R3 (cost): If requirements R1 and R2 are satisfied by multiple configurations, the UUV should use one of these configurations that minimises the cost function

$$cost = w_1 E + w_2 sp^{-1}, \tag{4.1}$$

where E is the energy used by the sensors to survey a 10m mission distance, and $w_1, w_2 > 0$ are weights that reflect the relative importance of carrying out the mission with reduced battery usage and completing the mission faster.[1]

R4 (safety): If a configuration that meets requirements **R1**–**R3** is not identified within 2 seconds after a sensor rate change, the UUV speed must be reduced to 0m/s. This ensures that the UUV does not advance more than the distance it can cover at its maximum speed within 2 seconds without taking appropriate measurements, and waits until the controller identifies a suitable configuration (e.g., after the UUV sensors recover) or new instructions are provided by a human operator.

4.3.2 Foreign Exchange Trading System

The service-based system from the area of foreign exchange trading is taken from our recent work in [88]. This system, which we anonymise as FX for confidentiality reasons, is used by an European foreign exchange brokerage company. The FX system implements the workflow shown in Fig. 4.6 and described below.

An FX customer (called a trader) can use the system in two operation modes. In the *expert* mode, FX executes a loop that analyses market activity, identifies patterns that satisfy the trader's objectives, and automatically carries out trades. Thus, the *Market watch* service extracts real-time exchange rates (bid/ask price) of selected currency pairs. This data is used by a *Technical analysis* service that evaluates the current trading conditions, predicts future price movement, and decides if the trader's objectives are: (i) "satisfied" (causing the invocation of an *Order* service to carry out a trade); (ii) "unsatisfied" (resulting in a new *Market watch* invocation); or (iii) "unsatisfied with high variance" (triggering an *Alarm* service invocation to notify the trader about discrepancies/opportunities not covered by the trading objectives). In the *normal* mode, FX assesses the economic

[1] Cost (or *utility*) functions that employ weights to combine several performance, reliability, resource use and other quality attributes of software—accounting for differences in attribute value ranges and relative importance—are extensively used in self-adaptive software systems (e.g. [37, 67, 84, 164, 201]).

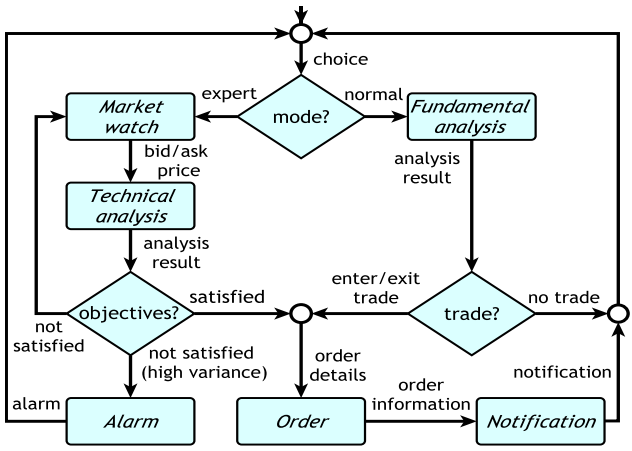


Figure 4.6: Foreign exchange trading (FX) workflow

outlook of a country using a *Fundamental analysis* service that collects, analyses and evaluates information such as news reports, economic data and political events, and provides an assessment on the country's currency. If satisfied with this assessment, the trader can use the *Order* service to sell or buy currency, in which case a *Notification* service confirms the completion of the trade. We assume that the FX system has to dynamically select third-party implementations for each service from Fig. 4.6, in order to meet the following system requirements:

R1 (reliability): Workflow executions must complete successfully with probability at least 0.9.

R2 (response time): The total service response time per workflow execution must be at most 5s.

R3 (cost): If requirements R1 and R2 are satisfied by multiple configurations, the FX system should use one of these configurations that minimises the cost function:

$$cost = w_1 price + w_2 time, \tag{4.2}$$

where $price$ and $time$ represent the total price of the services invoked by a workflow execution and the response time for a workflow execution, respectively, and $w_1, w_2 > 0$ are weights that encode the desired trade-off between price and response time.

R4 (safety): If a configuration that ensures requirements **R1**–**R3** cannot be identified within 2s after a change in service characteristics is signalled by the sensors of the self-adaptive FX system, the *Order* service invocation is bypassed, so that the FX system does not carry out any trade that might be based on incorrect or stale data.

Note that requirements R1–R3 express two constraints and an optimisation criterion that are qualitatively different from those specified by the requirements from our first case study (cf. Section 4.3.1). Nevertheless, our tool-supported instance

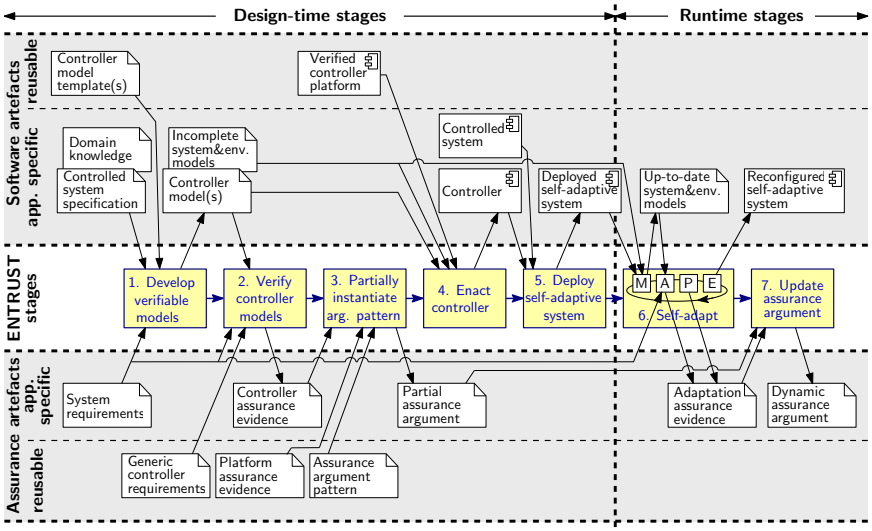


Figure 4.7: Stages and key artefacts of the ENTRUST methodology. In line with the two principles underpinning the methodology, its first stage involves the development of verifiable models for the controller, controlled system and environment of the self-adaptive system used throughout the remaining stages, and multiple stages reuse application-independent software and assurance artefacts.

of the ENTRUST methodology enabled the development of the self-adaptive FX system as described in Section 4.6.

4.4 The ENTRUST Methodology

The ENTRUST methodology supports the systematic engineering and assurance of self-adaptive systems based on monitor-analyse-plan-execute (MAPE) control loops. This is by far the most common type of control loop used to devise self-adaptive software systems [28, 67, 106, 125, 134, 135, 139, 164]. The engineering of self-adaptive systems based on essentially different control techniques, such as the control theoretical paradigm [172], as for example proposed in [75], is not supported by our methodology.

ENTRUST comprises the tool-supported design-time stages and the automated runtime stages shown in Fig. 4.7, and is underpinned by two key principles:

1) *Model-driven engineering is essential for developing trustworthy self-adaptive systems and their assurance cases.* As emphasised in the previous section, model-based analysis, simulation, testing and formal verification—at design time and during reconfiguration—represent the main sources of assurance evidence for self-adaptive software. As such, both the design-time and the runtime stages of our methodology are model driven. Models of the structure and behaviour of the functional components, controller and environment are the basis for the engineering and assurance of ENTRUST self-adaptive systems.

2) *Reuse of application-independent software and assurance artefacts significantly reduces the effort and expertise required to develop trustworthy self-adaptive systems.* Assembling an assurance case for a software system is a costly process that requires considerable effort and expertise. Therefore, the reuse of both software and assurance artefacts is essential for ENTRUST. In particular, the reuse of application-independent controller components and of templates for developing application-specific controller elements also enables the reuse of assurance evidence that these software artefacts are trustworthy.

The ENTRUST stages and their exploitation of these two principles are described in the remainder of this section.

4.4.1 Design-time ENTRUST Stages

4.4.1.1 Stage 1: Development of Verifiable Models

In ENTRUST, the engineering of a self-adaptive system with the architecture from Fig. 4.2 starts with the development of models for:

1) The controller of the self-adaptive system;
2) The relevant aspects of the controlled software system and its environment.

A combination of structural and behavioural models may be produced, depending on the evidence needed to assemble the assurance case for the self-adaptive system under development. ENTRUST is not prescriptive in this respect. However, we require that these models are *verifiable*, i.e., that they can be used in conjunction with methods such as model checking or simulation, to obtain evidence that the controller and the self-adaptive system meet their requirements. As an example, finite state transition models may be produced for the controllers of our UUV and FX systems from Section 4.3, enabling the use of model checking to verify that these controllers are deadlock free.

The verifiable models are application-specific. As illustrated in Fig. 4.7, their development requires *domain knowledge*,[1] is based on a *controlled system specification*, and is informed by the *system requirements*. As in other areas of software engineering, we envisage that tool-supported methods will typically be used to obtain these models. However, their manual development or fully automated synthesis are not precluded by ENTRUST.

In line with the "reuse of artefacts" principle, ENTRUST exploits the fact that the controllers of self-adaptive systems implement the established MAPE workflow, and uses application-independent *controller model template(s)* to devise the *controller model(s)*. These templates model the generic aspects of the MAPE workflow and contain placeholders for the application-specific elements of an ENTRUST controller.

Given the environmental and internal uncertainty that characterises self-adaptive systems, only *incomplete system and environment models* can be produced in this ENTRUST stage. These incomplete models may include unknown or estimated

[1] The ENTRUST software and assurance artefacts that appear in *italics* in the text are also shown in Fig. 4.7.

parameters, nondeterminism (i.e., alternative options whose likelihoods are unknown), parts that are missing, or some combination of all of these. For example, parametric Markov chains may be devised to enable the runtime analysis of the requirements for our UVV and FX systems detailed in Sections 4.3.1 and 4.3.2, respectively, by means of probabilistic model checking or simulation.

4.4.1.2 Stage 2: Verification of Controller Models

The main role of the second ENTRUST stage is to produce *controller assurance evidence*, i.e., compelling evidence that a controller based on the controller model(s) from Stage 1 will satisfy a set of *generic controller requirements*. These are requirements that must be satisfied in any self-adaptive system (e.g., deadlock freeness) and are predefined in a format compatible with that of the controller model templates and with the method that will be used to verify the controller models. For example, if labelled transition systems are used to model the controller and model checking to establish its correctness as in [63, 64], these generic controller requirements can be predefined as temporal logic formulae.

The controller assurance evidence must include evidence that the system requirements for application-specific failsafe operating mode(s) are always satisfied. In this way, a minimal assurance case is always available for the scenario when the runtime assurance evidence for other system requirements cannot be obtained and the self-adaptive system needs to switch to a degraded, failsafe mode of operation. Several fallback levels as proposed in [63] can also be supported in this way, with only the most degraded fallback level ensured through assurance evidence obtained in this ENTRUST stage. For example, requirements R4 of our UUV and FX systems from Section 4.3 specify failsafe operating modes for the two systems, so we will need to show that these requirements are always met.

The assurance evidence generated in this stage of the methodology may be obtained using a range of methods that include formal verification, theorem proving and simulation. The methods that can be used depend on the types of models produced in the previous ENTRUST stage, and on the generic controller requirements and system requirements for which assurance is sought. The availability of tool support in the form of model checkers, theorem provers, SMT solvers, domain-specific simulators, etc. will influence the choice of these methods.

Preparing the design-time models, i.e., developing verifiable models and verifying the controller models, comes with a cost. This cost can be reduced by using tool-supported methods and by exploiting reusable application-independent software, as done by the related approaches described in Section 4.8. Furthermore, these related approaches that only provide a fraction of the assurances that ENTRUST achieves (as detailed when we discuss related work in Section 4.8) operate with design-time models that require a comparable effort to specify the models and provide the controller assurance evidence.

4.4.1.3 Stage 3: Partial Instantiation of Assurance Argument Pattern

This ENTRUST stage uses the controller assurance evidence from Stage 2 to support the partial instantiation of a generic *assurance argument pattern* for self-adaptive software. As explained in Section 4.2.3, this pattern is an incomplete

assurance argument containing placeholders for the system-specific assurance evidence. A subset of the placeholders correspond to the controller assurance evidence obtained in Stage 2, and are therefore instantiated using this evidence. The result is a *partial assurance argument*, which still contains placeholders for the assurance evidence that cannot be obtained until the uncertainties associated with the self-adaptive system are resolved at runtime.

For example, the partial assurance argument for our UUV and FX systems should contain evidence that their controllers are deadlock free and that their fail-safe requirements R4 are always satisfied. These requirements can be verified at design time. In contrast, requirements R1–R3 for the two systems cannot be verified until runtime, when the controller acquires information about the measurement rates of the UUV sensors and the third-party services available for the FX operations, respectively. Assurance evidence that requirements R1–R3 are satisfied can only be obtained at runtime.

In addition to the two types of placeholders, the assurance argument pattern used as input for this stage includes assurance evidence that is application independent. In particular, it includes evidence about the correct operation of the *verified controller platform*, i.e. the software that implements application-independent controller functionality used to execute the ENTRUST controllers. This *platform assurance evidence* is reusable across self-adaptive systems.

4.4.1.4 Stage 4: Enactment of the Controller

This ENTRUST stage assembles the *controller* of the self-adaptive system. The process involves integrating the verified controller platform with the application-specific controller elements, and with the sensors and effectors that interface the controller with the controlled software system from Fig. 4.2.

The application-specific controller elements must be devised from the verified controller models, by using a trusted model-driven engineering method. This can be done using *model-to-text transformation*, a method that employs a trusted *model compiler* to generate a low-level executable representation of the controller models. Alternatively, the ENTRUST verified controller platform may include a trusted virtual machine[2] able to directly interpret and run the controller models. The second, *model interpretation* method [177], has the advantage that it eliminates the need to generate controller code and to provide additional assurances for it.

4.4.1.5 Stage 5: Deployment of the Self-Adaptive System

In the last design-time stage, the integrated controller and *controlled components* of the self-adaptive system are installed, preconfigured and activated by means of an application-specific process. The pre-configuration is responsible for setting the deployment-specific parameters and architectural aspects of the system. For example, the pre-configuration of the UUV system from Section 4.3.1 involves selecting the initial speed and active sensor set for the UUV, whereas for the FX system from Section 4.3.2 it involves choosing initial third-party implementations for each FX service.

[2] Throughout the paper, the term "virtual machine" refers to a software component capable to interpret and execute controller models, much like a Java virtual machine executes Java code.

The *deployed self-adaptive system* will be fully configured and a complete assurance argument will be available only after the first execution of the MAPE control loop. This execution is typically triggered by the system activation, to ensure that the newly deployed self-adaptive system takes into account the current state of its environment as described next.

4.4.2 Runtime ENTRUST Stages

4.4.2.1 Stage 6: Self-adaptation

In this ENTRUST stage, the deployed self-adaptive system is dynamically adjusting its parameters and architecture in line with observed internal and environmental changes. To this end, the controller executes a typical MAPE loop that monitors the system and its environment, using the information obtained in this way to resolve the "unknowns" from the incomplete system and environment models. The resulting *up-to-date system and environment models* enable the MAPE loop to analyse the system compliance with its requirements after changes, and to plan and execute suitable reconfigurations if necessary.

Whenever the MAPE loop produces a *reconfigured self-adaptive system*, its analysis and planning steps generate *adaptation assurance evidence* confirming the correctness of the analysis results and of the reconfiguration plan devised on the basis of these results. This assurance evidence is a by-product of analysis and planning methods that may include runtime verification, simulation and runtime model checking. Irrespective of the methods that produce it, the adaptation assurance evidence is essential for the development of a complete assurance argument in the next ENTRUST stage.

4.4.2.2 Stage 7: Synthesis of Dynamic Assurance Argument

The final ENTRUST stage uses the adaptation correctness evidence produced by the MAPE loop to fill in the placeholders from the partial assurance argument, and to devise the complete assurance case for the reconfigured self-adaptive system. For example, runtime evidence that requirements R1–R3 of the UUV and FX systems from Section 4.3 are met will be used to complete the remaining placeholders from their partial assurance arguments. Thus, an ENTRUST assurance case is underpinned by a *dynamic assurance argument* that is updated after each reconfiguration of the system parameters and architecture. This assurance case captures both the full assurance argument and the evidence that justifies the active configuration of the self-adaptive system.

The ENTRUST assurance case versions generated for every system reconfiguration have two key uses. First, they allow decision makers and auditors to understand and assess the present and past versions of the assurance case. Second, they allow human operators to endorse major reconfiguration plans in human-supervised self-adaptive systems. This type of self-adaptive systems is of particular interest in domains where human supervision represents an important risk mitigation factor or may be required by regulations. As an example, UK Civil Aviation Authority regulations [191] permit self-adaptation in certain functions (e.g.,

power management, flight management and collision avoidance) of unmanned aircraft of no more than 20 kg provided that the aircraft operates within the visual line of sight of a human operator.

4.5 Tool-Supported Instance of ENTRUST

This section presents an instance of ENTRUST in which the stages described in Section 4.4 are supported by the modelling and verification tools UPPAAL [17] and PRISM [126]. We start with an overview of this tool-supported ENTRUST instance in Section 4.5.1, followed by a description of each of its stages in Section 4.5.2.

4.5.1 Overview

The ENTRUST methodology can be used with different combinations of modelling, verification and controller enactment methods, which may employ different self-adaptive system architectures and types of assurance evidence. This section presents a tool-supported instance of ENTRUST that uses one such combination of methods. We developed this instance of the methodology with the aim to validate ENTRUST and to ease its adoption.

Our ENTRUST instance supports the engineering of self-adaptive systems with the architecture shown in Fig. 4.8. The reusable verified controller platform at the core of this architecture comprises:

1) A *Trusted Virtual Machine* that directly interprets and executes models of the four steps from the MAPE control loop[1] (i.e., the ENTRUST controller models).
2) A *Probabilistic Verification Engine* that is used to verify stochastic models of the controlled system and its environment during the analysis step of the MAPE loop.

Using the *Trusted Virtual Machine* for controller model interpretation eliminates the need for a model-to-text transformation of the controller models into executable code, which is a complex, error-prone operation. Not having to devise this transformation and to provide assurance evidence for it are major benefits of our ENTRUST instance. Although we still need assurance evidence for the virtual machine, this was obtained when we developed and verified the virtual machine,[2] and is part of the reusable *platform assurance evidence* for the ENTRUST instance.

The *Probabilistic Verification Engine* consists of the verification libraries of the probabilistic model checker PRISM [126] and is used by the analysis step of the MAPE control loop. As such, our ENTRUST instance works with:

1) Stochastic finite state transition models of the controlled system and the environment, defined in the PRISM high-level modelling language. Incomplete

[1] Hence the controller models are depicted as software components in Fig. 4.8.

[2] This assurance evidence is in the form of a comprehensive test suite and a report describing its successful execution by the virtual machine, both of which are available on our ENTRUST project website at `https://www-users.cs.york.ac.uk/simos/ENTRUST/`.

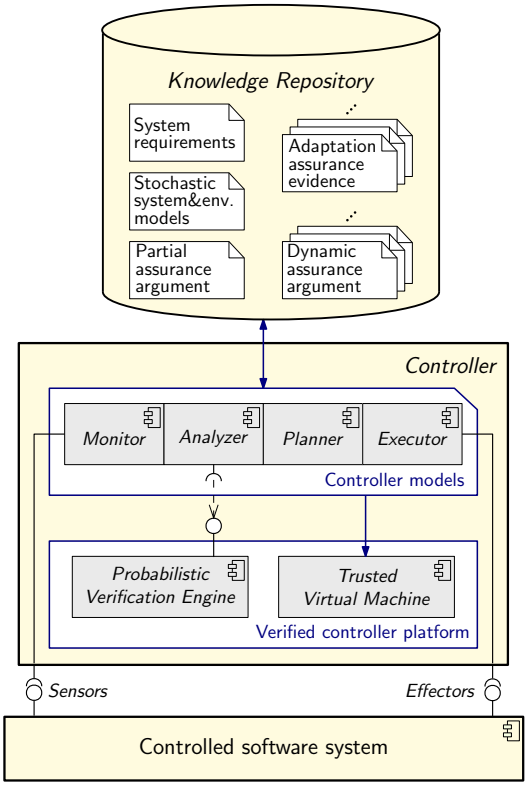


Figure 4.8: Architecture of an ENTRUST self-adaptive system

versions of these models are devised in Stage 1 of ENTRUST, and have their unknowns resolved at runtime. All types of models that PRISM can analyse are supported, including discrete- and continuous-time Markov chains (DTMCs and CTMCs), Markov decision processes (MDPs) and probabilistic automata (PAs).

2) Runtime-assured system requirements expressed in the appropriate variant of probabilistic temporal logic, i.e., probabilistic computation tree logic (PCTL) for DTMCs, MDPs and PAs, and continuous stochastic logic (CSL) for CTMCs.

This makes our instantiation of the generic ENTRUST methodology applicable to self-adaptive systems whose non-functional (e.g., reliability, performance, resource usage and cost-related) requirements can be specified in the above logics, and whose behaviour related to these requirements can be described using stochastic models. As shown by the recent work of multiple research groups (e.g., [37, 38, 39, 41, 68, 71, 76, 158, 180]), this represents a broad and important class of self-adaptive software that includes a wide range of service-based systems, web applications, resource management systems, and embedded systems.

Also developed in Stage 1 of ENTRUST, the four controller models form an application-specific network of interacting timed automata [4], and are expressed in the modelling language of the UPPAAL verification tool suite [17].

Accordingly, UPPAAL is used in Stage 2 of ENTRUST to verify the compliance of the controller models with the generic controller requirements and with any system requirements that can be assured at design time. These requirements are defined in computation tree logic (CTL) [52].

In Stage 3 of our ENTRUST instance, a partial assurance argument is devised starting from an assurance argument pattern represented in *goal structuring notation* (GSN) [115].

The controller enactment from Stage 4 involves integrating the timed-automata controller models with our verified controller platform.

In Stage 5 of ENTRUST, the controlled software system and its enacted controller are deployed, together with a *Knowledge Repository* that supports the operation of the controller. Initially, this repository contains: (i) the partial assurance argument from Stage 3; (ii) the system requirements to be assured at runtime; and (iii) the (incomplete) stochastic system and environment models from Stage 1.

During the execution of the MAPE loop in Stage 6 of ENTRUST, the *Monitor* obtains information about the system and its environment through *Sensors*. This information is used to resolve the unknowns from the stochastic models of the controlled system and its environment. Examples of such unknowns include probabilities of transition to 'failure' states for a DTMC, MDP or PA, rates of transition to 'success' states for a CTMC, and sets of states and transitions modelling certain system behaviours. After each update of the stochastic system and environment models, the *Analyzer* reverifies the compliance of the self-adaptive system with its runtime-assured requirements. When the requirements are no longer met, the *Analyzer* uses the verification results to identify a new system configuration that restores this compliance, or to find out that such a configuration does not exist and to select a predefined failsafe configuration. The step-by-step actions needed to achieve the new configuration are then established by the *Planner* and implemented by the *Executor* through the *Effectors* of the controlled system.

Using the *Probabilistic Verification Engine* enables the *Analyzer* and *Planner* to produce assurance evidence justifying their selection of new configurations and of plans for transitioning the system to these configurations, respectively. This adaptation assurance evidence is used to synthesise a fully-fledged, dynamic GSN assurance argument in Stage 7 of our ENTRUST instance. As indicated in Fig. 4.8, versions of the adaptation assurance evidence and of the dynamic assurance argument justifying each reconfiguration of the self-adaptive system are stored in the *Knowledge Repository*.

The implementation of the ENTRUST stages in our tool-supported instance of the methodology is summarised in Table 4.2 and described in further detail in Section 4.5.2.

Table 4.2: Stages of the tool-supported instance of the ENTRUST methodology

Stage	Type	Description	Supporting tool(s)
1	tool supported	Timed automata controller models developed from UPPAAL templates	UPPAAL
		Incomplete stochastic models of the controlled system and environment developed based on system specification and domain knowledge	PRISM
2	tool supported	Controller models verified to obtain controller assurance evidence	UPPAAL
3	manual	Partial assurance argument devised from GSN assurance argument pattern	–
4	manual	Controller enacted by integrating the verified controller models and platform	–
5	manual	Controlled system, controller and knowledge repository deployed	–
6	automated	MAPE control loop continually executed to ensure the system requirements	PRISM & ENTRUST controller platform
7	automated	GSN dynamic assurance argument generated	ENTRUST controller platform

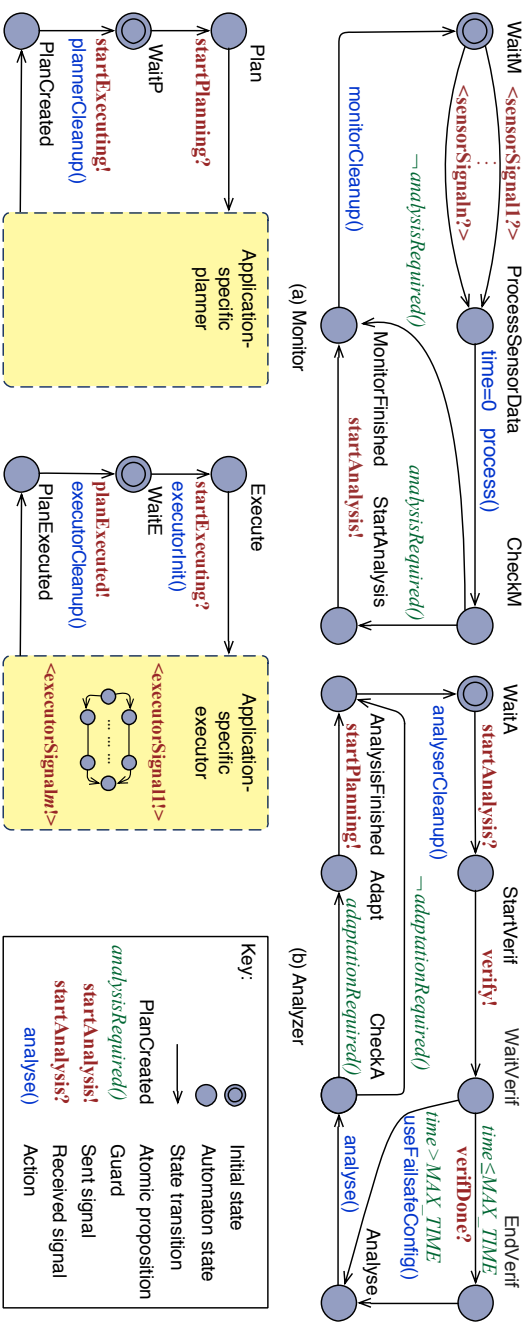


Figure 4.9: Event-triggered MAPE model templates

4.5.2 Stage Descriptions

4.5.2.1 Development of Verifiable Models

Controller models. We devised two types of templates for the four controller models from Fig. 4.8: (i) *event triggered*, in which the monitor automaton is activated by a sensor-generated signal indicating a change in the managed system or the environment; and (ii) *time triggered*, in which the monitor is activated periodically by an internal clock. The event-triggered automaton templates are shown in Fig. 4.9 using the following font and text style conventions:

- Sans-serif font is used to annotate states with the atomic propositions (i.e. boolean properties) that hold in those states, e.g. PlanCreated from the Planner automaton;
- *Italics text* is used for the guards that annotate state transitions with the conditions which must hold for the transitions to occur, e.g. *time*$\leq$*MAX_TIME* from the Analyzer automaton;
- State transitions are additionally annotated with the actions executed upon taking the transitions, and these actions are also shown in sans-serif font, e.g. time=0 to initialise a timer in the Monitor automaton;
- **Bold text** is used for the synchronisation channels between two automata—these channels are specified as pairs comprising a '!'-decorated sent signal and a '?'-decorated received signal with the same name, e.g., **startAnalysis!** and **startAnalysis?** from the monitor and analyzer automata, respectively. The two transitions associated with a synchronisation channel can only be taken at the same time.

Finally, signals in angle brackets '$\langle\rangle$' are placeholders for application-specific signal names, and guards and actions decorated with brackets '()' represent application-specific C-style functions.

To specialise these model templates for a particular system and application, software engineers need: (a) to replace the signal placeholders with real signal names; (b) to define the guard and action functions; and (c) to devise the automaton regions shaded in Fig. 4.9. For example, for the monitor automaton the engineers first need to replace the placeholders $\langle$**sensorSignal**$_1$**?**$\rangle$, ..., $\langle$**sensorSignal**$_n$**?**$\rangle$ with sensor signals announcing relevant changes in the managed system. They must then implement the functions process(), *analysisRequired()* and monitorCleanup(), whose roles are to process the sensor data, to decide if the change specified by this data requires the "invocation" of the analyzer through the **startAnalysis!** signal, and to carry out any cleanup that may be required, respectively. Details about the other automata from Fig. 4.9 are available on our project website, which also provides implementations of these MAPE model templates in the modelling language of the UPPAAL verification tool suite [17].

Parametric stochastic models. These models used by the ENTRUST control loop at runtime are application specific, and need to be developed from scratch. Their parameters correspond to probabilities or rates of transition between model states, and are continually estimated at runtime, based on change information provided

Table 4.3: Stochastic models supported by the ENTRUST instance, with citations of representative research that uses them in self-adaptive systems

Type of stochastic model	Non-functional requirement specification logic
Discrete-time Markov chains [36, 37, 68, 71, 72, 92]	PCTL[a], LTL[b], PCTL*[c]
Markov decision processes [76]	PCTL[a], LTL[b], PCTL*[c]
Probabilistic automata [38, 113]	PCTL[a], LTL[b], PCTL*[c]
Continuous-time Markov chains [34, 35, 87]	CSL[d]
Stochastic games [41, 42]	rPATL[e]

[a] Probabilistic Computation Tree Logic [21, 96]

[b] Linear Temporal Logic [155]

[c] PCTL* is a superset of PCTL and LTL

[d] Continuous Stochastic Logic [10, 12]

[e] reward-extended Probabilistic Alternating-time Temporal Logic [46]

by the sensors of the controlled system. As such, the verification of these models at runtime enables the ENTRUST analyzer to identify configurations it can use to meet the system requirements after unexpected changes, as described in detail in [35, 37, 38, 68, 72]. The types of stochastic models supported by our ENTRUST instance are shown in Table 4.3. As illustrated by the research work cited in the table, the temporal logics used to express the properties of these models support the specification of numerous performance, reliability, safety, resource usage and other non-functional requirements that recent surveys propose for self-adaptive systems [49, 199].

To ensure the accuracy of the stochastic models described above, ENTRUST can rely on recent advances in devising these models from logs [92, 153] and UML activity diagrams [31, 80], and in dynamically and accurately updating their parameters based on sensor-provided runtime observations of the controlled system [32, 36, 68, 74].

4.5.2.2 Verification of Controller Models

During this ENTRUST stage, a trusted model checker is used to verify the network of MAPE automata devised in the previous section. This verification yields evidence that the MAPE models satisfies a set of key safety and liveness properties that include both generic and application-specific properties. Table 4.4 shows a non-exhaustive list of generic properties that we assembled for the current version of ENTRUST. Although these properties are application-independent, verifying that an ENTRUST controller satisfies them is possible only after its application-specific MAPE models were devised. This involves completing the application-specific parts of the planner and executor automata, and implementing the functions for the *guards* and actions from all the model templates.

Additionally, automata that simulate the controller sensors, runtime probabilistic verification engine and effectors from Fig. 4.8 need to be defined to enable this verification. The sensors, verification engine and effectors automata have to synchronise with the relevant monitor, analyzer and executor signals, respectively. The sensors automaton and verification automaton also have to exercise the possible paths through the monitor, analyzer and planner automata (and indirectly the executor automaton). To this end, they can nondeterministically populate the knowledge repository with data that satisfies all the different guard combinations. Alternatively, a finite collection of the two automata can be used to verify subsets of all possible MAPE paths, as long as the union of all such subsets covers the entire behaviour space of the MAPE network of automata.

Note that these application-specific elements of the MAPE automata are much larger than the application-independent elements from the MAPE model templates. Therefore, we do not use compositional model checking [54, 113] to verify the two parts of the MAPE automata separately, with the application-independent elements verified once and for all. Such an approach would increase the complexity of the verification task (e.g. by requiring the identification and verification of less intuitive "assumptions" [55] that the application-specific parts of the automata need to "guarantee") without any noticeable reduction in the verification time, almost all of which would be required to verify the application-specific automata elements.

4.5.2.3 Partial Instantiation of Assurance Argument Pattern

We used the *Goal Structuring Notation* (GSN) introduced in Section 4.2.2 to devise a reusable *assurance argument pattern* (cf. Section 4.2.3) for self-adaptive software. Unlike all existing assurance argument patterns [101], our new pattern captures the fact that for self-adaptive software the assurance process cannot be completed at design time. Instead, it is a continual process where some design features and code elements are dynamically reconfigured and executed during self-adaptation. As such, the detailed claims and evidence for meeting the system requirements must vary with self-adaption, and thus ENTRUST assurance cases must evolve dynamically at runtime.

The ENTRUST assurance argument pattern is shown in Fig. 4.10. Its root goal, **ReqsSatisfied**, states that the system requirements are satisfied at all times. These requirements are typically allocated to the software from the higher-level system analysis process, so the justifications of their derivation, validity and completeness are addressed as part of the overall system assurance case (which is outside the scope of the software assurance case). **ReqsSatisfied** is supported by a sub-claim based on (i.e. in the context of) the current configuration (**ReqsConfiguration**) and by a reconfiguration sub-claim (**Reconfig**). That is, the pattern shows that we are guaranteeing that the current configuration satisfies the requirements (in the absence of changes) and that the ENTRUST controller will plan and execute a reconfiguration that will satisfy these requirements (should a change occur).

The pattern justifies how the system requirements are achieved for each configuration by using a sub-goal **RxAchieved** for each requirement Rx. Further, a new configuration has the potential to introduce erroneous behaviours (e.g., dead-

Table 4.4: Generic properties that should be satisfied by an ENTRUST controller

ID	Informal description	Specification in computation tree logic (CTL) [52]
P1	The ENTRUST controller is deadlock free.	A□ not deadlock
P2	Whenever analysis is required, the Analyser eventually carries out this action.	A□ (Monitor.StartAnalysis → A◇ Analyzer.Analyse)
P3	Whenever the system requirements are violated, a stepwise reconfiguration plan is eventually assembled.	A□ (Analyzer.Adapt → A◇ Planner.PlanCreated)
P4	Whenever a stepwise plan is assembled, the Executor eventually implements it.	A□ (Planner.PlanCreated → A◇ Executor.PlanExecuted)
P5	Whenever the Monitor starts processing the received data, it eventually terminates its execution.	A□ (Monitor.ProcessSensorData → A◇ Monitor.Finished)
P6	Whenever the Analyser begins the analysis, it eventually terminates its execution.	A□ (Analyzer.Analyse → A◇ Analyzer.AnalaysisFinished)
P7	A plan is eventually created, each time the Planner starts planning.	A□ (Planner.Plan → A◇ Planner.PlanCreated)
P8	Whenever the Executor starts executing a plan, the plan is eventually executed.	A□ (Executor.Execute → A◇ Executor.PlanExecuted)
P9	Whenever adaptation is required, the current configuration and the best configuration differ.	A□ (Analyzer.Adapt → currentConfig != newConfig)

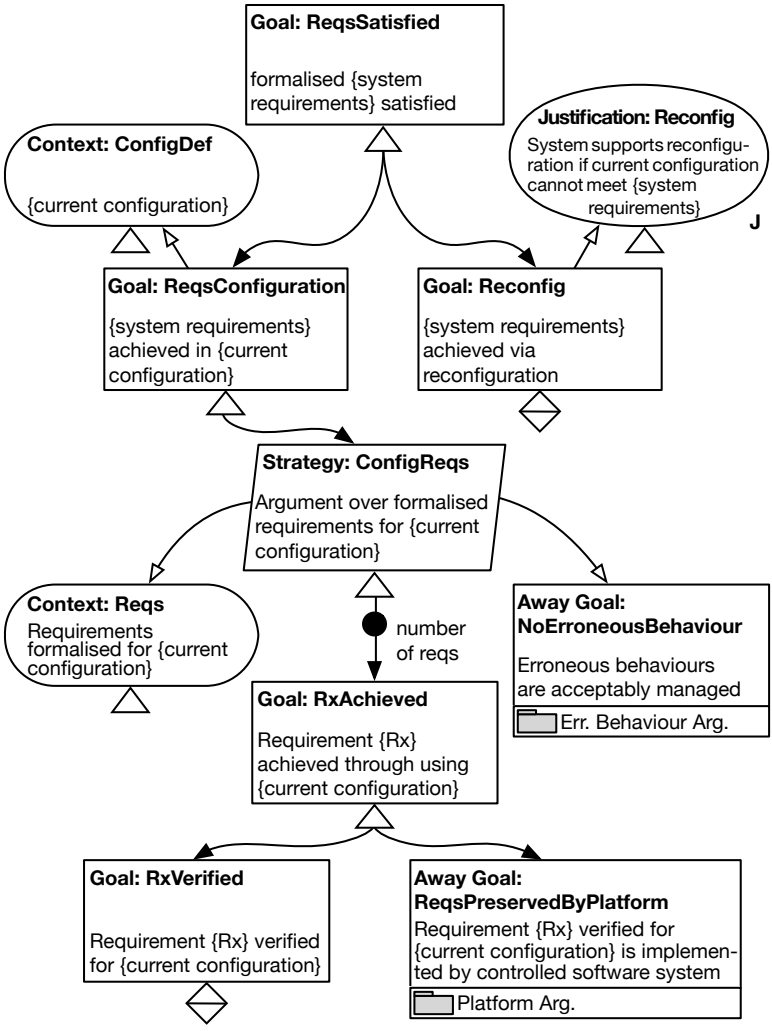


Figure 4.10: ENTRUST assurance argument pattern.

locks). The justification for the absence of these errors is provided via the away goal **NoErroneousBehaviour** (described below). The pattern concludes with the goals **RxVerified** and **ReqsPreservedByPlatform**, which justify the verification and the implementation of the formalised requirements, respectively. The away goal **ReqsPreservedByPlatform** confirms that the controlled system handles correctly the reconfiguration commands received through effectors. This away goal is obtained using standard assurance processes, which are outside the scope of this paper.

As shown Fig. 4.11, the **NoErroneousBehaviour** away goal is supported by two sub-claims. The **FMsManaged** sub-claim uses the goals **FMsIdentified** and **ReqsDerived** to state that the relevant "failure modes" for the self-adaptive system

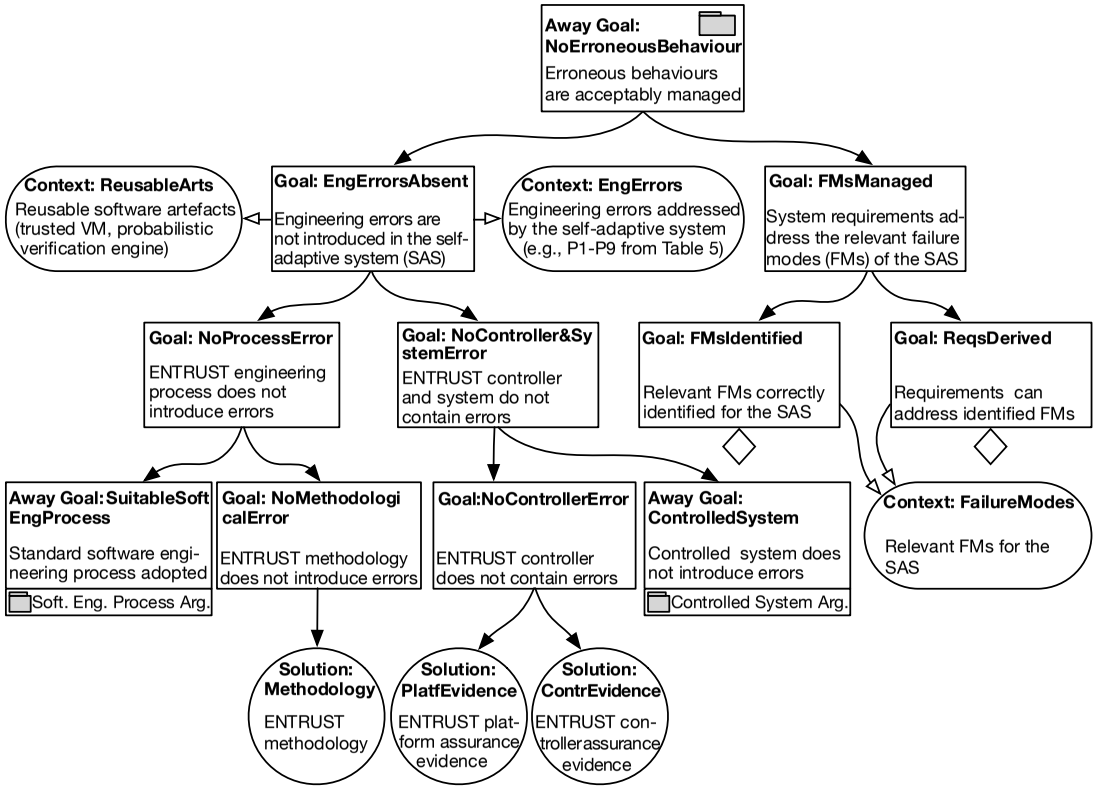


Figure 4.11: Away goal **NoErroneousBehaviour**, which justifies the absence of errors due to reconfiguration and is based on the existing GSN pattern Hazardous Contribution Software Safety Argument from the existing GSN catalogue [101]

have been identified and that the system requirements fully address these failure modes. We leave the two goals undeveloped, as they are achieved using standard requirements engineering and assurance practices. The **EngErrorsAbsent** sub-claim states that the engineering of the self-adaptive system does not introduce errors in the context of the ENTRUST reusable artefacts (i.e., of our trusted virtual machine and probabilistic verification engine) and of the generic properties that an ENTRUST controller has to satisfy. **EngErrorsAbsent** is in turn supported by two sub-goals, **NoProcessError** and **NoController&SystemError**. The former sub-goal is obtained through using suitable software engineering processes (via the away goal **SuitableSoftEngProcess**, which also covers the use of the methods mentioned in Section 4.5.2.1 to ensure the accuracy of the ENTRUST stochastic models) and through avoiding methodological errors by using the ENTRUST methodology. The latter sub-goal, **NoController&SystemError**, is achieved by claims about:

1) The absence of controller errors. This is supported by (i) the controller verification evidence from Stage 2 of ENTRUST; and (ii) the reusable platform assurance evidence, which includes (testing) evidence about the correct operation of the model checkers UPPAAL and PRISM, based on their long track record of successful adoption across multiple domains and on our own experience of using them to develop self-adaptive systems.
2) The absence of controlled system errors, covered by the **ControlledSystem** away goal.

The away goals **SuitableSoftEngProcess** and **ControlledSystem** are obtained following existing software assurances processes, and thus we do not describe them here.

The partial instantiation of the assurance argument pattern in the last design-time stage of ENTRUST produces a *partially-developed and partially-instantiated* assurance argument [61]. This includes placeholders for items of evidence that can only be instantiated and developed based on operational data, i.e., the runtime verification evidence that is generated by the analysis and planning steps of the ENTRUST controller.

4.5.2.4 Enactment of the Controller

In this stage, the controller from Fig. 4.8 is assembled by integrating the MAPE controller models discussed in Section 4.5.2.1, the ENTRUST verified controller platform and application-specific sensor, effector and stochastic model management components. The application-specific components include generic functionality such as the signals through which these components synchronise with the MAPE automata (e.g., **verify?** and **planExecuted?**). Accordingly, our current version of ENTRUST includes abstract Java classes that provide this common functionality. These abstract classes, which we made available on the project website, need to be specialised for each application. Thus, the specialised sensors and effectors must use the APIs of the managed software system to observe its state and environment, and to modify its configuration, respectively. The stochastic model management component must specialise the probabilistic verification engine so

that it instantiates the parametric stochastic models using the actual values of the managed system and environment parameters (provided by sensors) and analyses the application-specific requirements.

4.5.2.5 Deployment of the Self-Adaptive System

As explained in Section 4.4.1.5, the role of this stage is to integrate the ENTRUST controller and the controlled software system into a self-adaptive software system that is then installed, preconfigured and set running. In particular, the preconfiguration must select initial values for all the parameters of the controlled system. Immediately after it starts running and until the first execution of the MAPE control loop, the system functions as a traditional, non-adaptive software system. As such, a separate assurance argument (which is outside the scope of this paper) must be developed using traditional assurance methods, to confirm that the initial system configuration is suitable.

The newly running software starts to behave like a self-adaptive system with the first execution of the MAPE control loop, as described in the next two sections.

4.5.2.6 Self-Adaptation

In this ENTRUST stage, the deployed self-adaptive system is dynamically adjusting its configuration in line with the observed internal and environmental changes. The use of continual verification within the ENTRUST control loop produces assurance evidence that underpins the dynamic generation of assurance cases in the next stage of our ENTRUST instance.

4.5.2.7 Synthesis of Dynamic Assurance Argument

The ENTRUST assurance case evolves in response to the results of the MAPE process, e.g., *time-triggered* and *event-triggered* outputs of the monitor, the outcomes of the analyzer, the mitigation actions developed by the planner and their realisation by the executor. This offers a dynamic approach to assurance because the full instantiation of the ENTRUST assurance argument pattern is left to runtime, i.e. the only stage when the evidence required to complete the argument becomes available. As such, the assurance case resulting from this stage captures the full argument and evidence for the justification of the current configuration of the self-adaptive system.

4.6 Applying the ENTRUST Methodology

4.6.1 Development of Verifiable Models

4.6.1.1 UUV System

Controller models. We instantiated the ENTRUST model templates for the UUV system from Section 4.3.1, obtaining the automata shown in Fig. 4.12. The signal **newRate?** is the only sensor signal that the monitor automaton needs to deal with, by reading a new UUV-sensor measurement rate (in process()) and checking whether this rate has changed to such extent that a new analysis is required (in

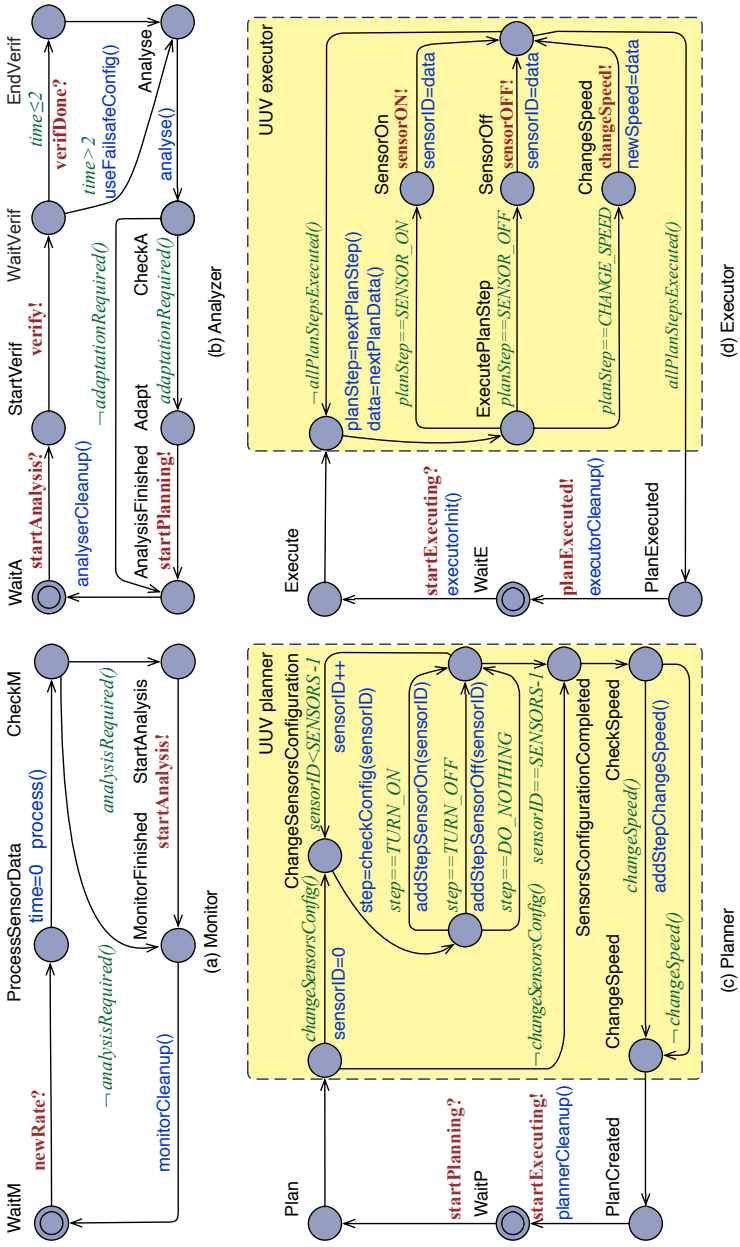


Figure 4.12: UUV MAPE automata that instantiate the event-triggered ENTRUST model templates

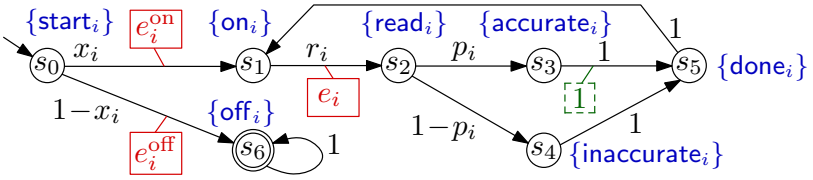


Figure 4.13: CTMC model M_i of the i-th UUV sensor, adopted from [87]

analysisRequired()). If analysis is required, the analyzer automaton sends a **verify!** signal to invoke the runtime verification engine, and thus verifies which UUV configurations satisfy requirements R1 and R2 and with what *cost*. The function analyse() uses the verification results to select a configuration that satisfies R1 and R2 with minimum *cost* (cf. requirement R3). If no such configuration exists or the verification does not complete within 2 seconds and the guard '*time>2*' is triggered, a zero-speed configuration is selected (cf. requirement R4). If the selected configuration is not the one in use, *adaptationRequired()* returns true and the **startPlanning!** signal is sent to initiate the execution of the planner automaton. The planner assembles a stepwise plan for changing to the new configuration by first switching on any UUV sensors that require activation, then switching off those that are no longer needed, and finally adjusting the UUV speed. These reconfiguration steps are carried out by the executor automaton by means of **sensorON!**, **sensorOFF!** and **changeSpeed!** signals handled by the effectors from Fig. 4.8, as described in Section 4.5.2.4.

Parametric stochastic models. Fig. 4.13 shows the CTMC model M_i of the i-th UUV sensor. From the initial state s_0, the system transitions to state s_1 or s_6 if the sensor is switched on ($x_i = 1$) or off ($x_i = 0$), respectively. The sensor takes measurements with rate r_i, as indicated by the transition $s_1 \rightarrow s_2$. A measurement is accurate with probability p_i as shown by the transition $s_2 \rightarrow s_3$; when inaccurate, the transition $s_2 \rightarrow s_4$ is taken. While the sensor is active this operation is repeated, as modelled by the transition $s_5 \rightarrow s_1$. The model is augmented with two *reward structures*. A "measure" structure, shown in a dashed rectangular box, associates a reward of 1 to each accurate measurement taken. An "energy" structure, shown in solid rectangular boxes, associates the energy used to switch the sensor on (e_i^{on}) and off (e_i^{off}) and to perform a measurement (e_i) with the transitions modelling these events. The model M of the n-sensor UUV is given by the parallel composition of the n sensor models: $M = M_1||...||M_n$; and the QoS system requirements are specified using CSL as follows:

R1: $R^{\text{measure}}_{\geq 20}[C^{\leq 10/sp}]$

R2: $R^{\text{energy}}_{\leq 120}[C^{\leq 10/sp}]$

R3: $\text{minimise}\,(w_1 E + w_2 sp^{-1})$, where $E = R^{\text{energy}}_{=?}[C^{\leq 10/sp}]$

where $10/sp$ is the time taken to travel 10m at speed sp. As requirement R4 is a failsafe requirement, we verify it at design time as explained in Section 4.6.2.1, so it is not encoded into CSL.

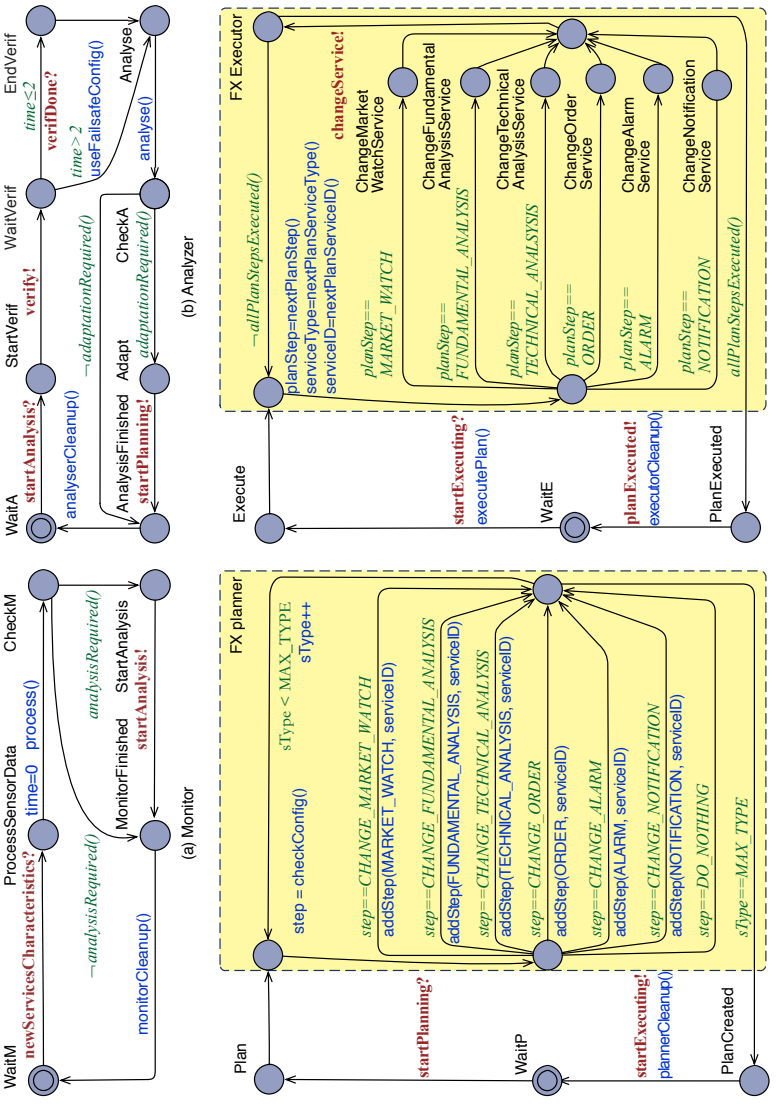


Figure 4.14: FX MAPE automata that instantiate the event-triggered ENTRUST model templates

4.6.1.2 FX System

Controller models. We specialised our event-triggered MAPE model templates for the FX system. The resulting MAPE models are shown in Fig. 4.14, where the shaded areas in Planner and Executor automata indicate the FX-specific steps for assembling a plan and executing the adaptation, respectively. The implementations of all *guards* and actions decorated with brackets '()' (which represent application-specific C-style functions, as explained in Section 4.5.2.1) are available on our project website.

Parametric stochastic models. To model the runtime behaviour of the FX system, we used the parametric discrete-time Markov chain (DTMC) depicted in Fig. 4.15. In this DTMC, constant transition probabilities derived from system logs are associated with the branches of the FX workflow from Fig. 4.6. In contrast, state transitions that model the success or failure of service invocations are associated with parameterised probabilities, which are unknown until the runtime selection of the FX services. Likewise, the "price" and (response) "time" reward structures (shown in solid and dashed boxes, respectively) are parametric and depend on the combination of FX services dynamically selected by the ENTRUST controller.

Finally, we formalised requirements R1–R3 in rewards-augmented probabilistic computational tree logic (PCTL):

R1: $P_{\geq 0.9}[F\,\mathsf{done}]$

R2: $R^{\mathrm{time}}_{\leq 5}[F\,\mathsf{done}]$

R3: $\mathrm{minimise}\,(w_1 price + w_2 time)$, where
$price = R^{\mathrm{price}}_{=?}[F\,\mathsf{done}]$ and $time = R^{\mathrm{time}}_{=?}[F\,\mathsf{done}]$

4.6.1.3 Discussion

The ENTRUST controller model templates supported the development of the UUV and FX controller models with structural changes confined to the Planner and Executor automata. Despite the differences between the sensor data used by the two systems (cf. Table 4.1), the Monitor and Analyzer automata could be instantiated with all application-specific functionality provided by the guard and action functions associated with the automata transitions. Different types of stochastic models were required for the two systems (continuous time for the UUV system, and discrete time for the FX system) as the differences in their requirements and uncertainties needed the modelling of different aspects of their behaviour.

4.6.2 Verification of Controller Models

4.6.2.1 UUV System

We used the UPPAAL model checker [17] to verify that the network of MAPE automata from Fig. 4.12 (which we made available on our project website) satisfies all the generic correctness properties from Table 4.4, as well as the application-specific property

R4: A□ (Analyzer.Analyse ∧ Analyzer.time>2 →
A◇ Planner.Plan ∧ newConfig.speed==0),

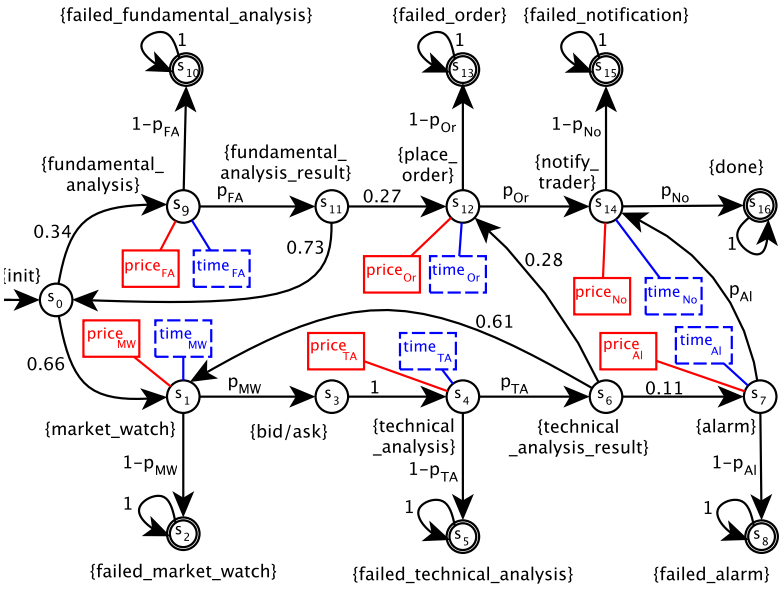


Figure 4.15: Parametric DTMC model of the FX system; p_{MW}, p_{TA}, ..., $time_{\mathsf{MW}}$, $time_{\mathsf{TA}}$, ..., and $price_{\mathsf{MW}}$, $price_{\mathsf{TA}}$, ..., represent the *reliability* (i.e. success probability), the *response time* and the *price*, respectively, of the implementations used for the MW, TA, ... system services.

which represents the CTL encoding of requirement R4. To carry out this verification, we defined simple sensors, verification engine and effectors automata as described above. We used a simple one-state effectors automaton with transitions returning to its single state for each of the received signals **sensorON?**, **sensorOFF? changeSpeed?** and **planExecuted?**; and a finite collection of sensor–verification engine automata pairs that together exercised all possible paths of the MAPE automata from Fig. 4.12. These auxiliary UPPAAL automata are available on the project website.

4.6.2.2 FX System

We used the model checker UPPAAL to verify that the MAPE automata network from Fig. 4.14 satisfies the generic controller correctness properties in Table 4.4, and a FX-specific CSL property corresponding to the failsafe requirement R4 of the FX system:

R4: A□ (Analyzer.Analyse ∧ Analyzer.time>2 →
A◇ Planner.Plan ∧ newConfig.Order==NoSvc),

where 'newConfig.Order==NoSvc' signifies that no service is used to implement the *Order* operation (i.e., the operation is skipped).

4.6.2.3 Discussion

The availability of a set of generic properties that must be satisfied by all ENTRUST controllers (cf. Table 4.4) meant that an additional CSL property was only needed for the application-specific failsafe requirement. For both systems, this

additional property corresponds to the scenario where a suitable new configuration cannot be obtained timely, suggesting that using a property template may be feasible for this and potentially for other types of failsafe requirements.

4.6.3 Partial Instantiation of Assurance Argument Pattern

4.6.3.1 UUV System

Fig. 4.16 shows the partially-instantiated assurance argument pattern for the self-adaptive UUV system, in which we shaded the (partially) instantiated GSN elements. To keep the diagram clear, we only show the expansion for requirements R1 and R4, leaving R2 and R3 undeveloped. The goal **R1Achieved** (which needs to be further instantiated when the system configuration is dynamically selected) is supported by: (a) sub-claim **R1Verified**, whose associated solution placeholder **R1Result** remains uninstantiated and should constantly be updated by the ENTRUST controller at runtime; and (b) the away goal **ReqsPreservedByPlatform** described earlier in this section. The undeveloped and partially instantiated goals **R2Achieved** and **R3Achieved** have the same structure as **R1Achieved**. In contrast, the (failsafe) goal **R4Achieved** is fully instantiated because the solution **R4Result**, comprising UPPAAL verification evidence that R4 is achieved irrespective of the configuration of the self-adaptive system, was obtained in the second ENTRUST stage (verification of controller models), cf. Section 4.6.2.1.

4.6.3.2 FX System

We partially instantiated the ENTRUST assurance argument pattern for our self-adaptive FX system, as shown in Fig. 4.17.

4.6.3.3 Discussion

As shown in Figs. 4.16 and 4.17, roughly the top half of the partially instantiated assurance argument pattern comes from ENTRUST assurance pattern in Fig. 4.10. This part of the assurance argument captures assurance elements generic to all self-adaptive systems, allowing the developers of a self-adaptive system to focus on the application-specific elements, which they are often more familiar with.

4.6.4 Enactment of the Controller

4.6.4.1 UUV System

To assemble an ENTRUST controller for the UUV system, we implemented Java classes that extend the functionality of the abstract Sensors, Effectors and VerificationEngine classes from the ENTRUST distribution. In addition to synchronising with the relevant application-specific signals from the MAPE automata (e.g., **newRate?**), the specialised sensors and effectors invoke the relevant API methods of our UUV simulator. The specialised verification engine instantiates the parametric sensor models M_i from Fig. 4.13, $1 \leq i \leq n$, and verifies the CSL-encoded requirements from Section 4.6.1.1.

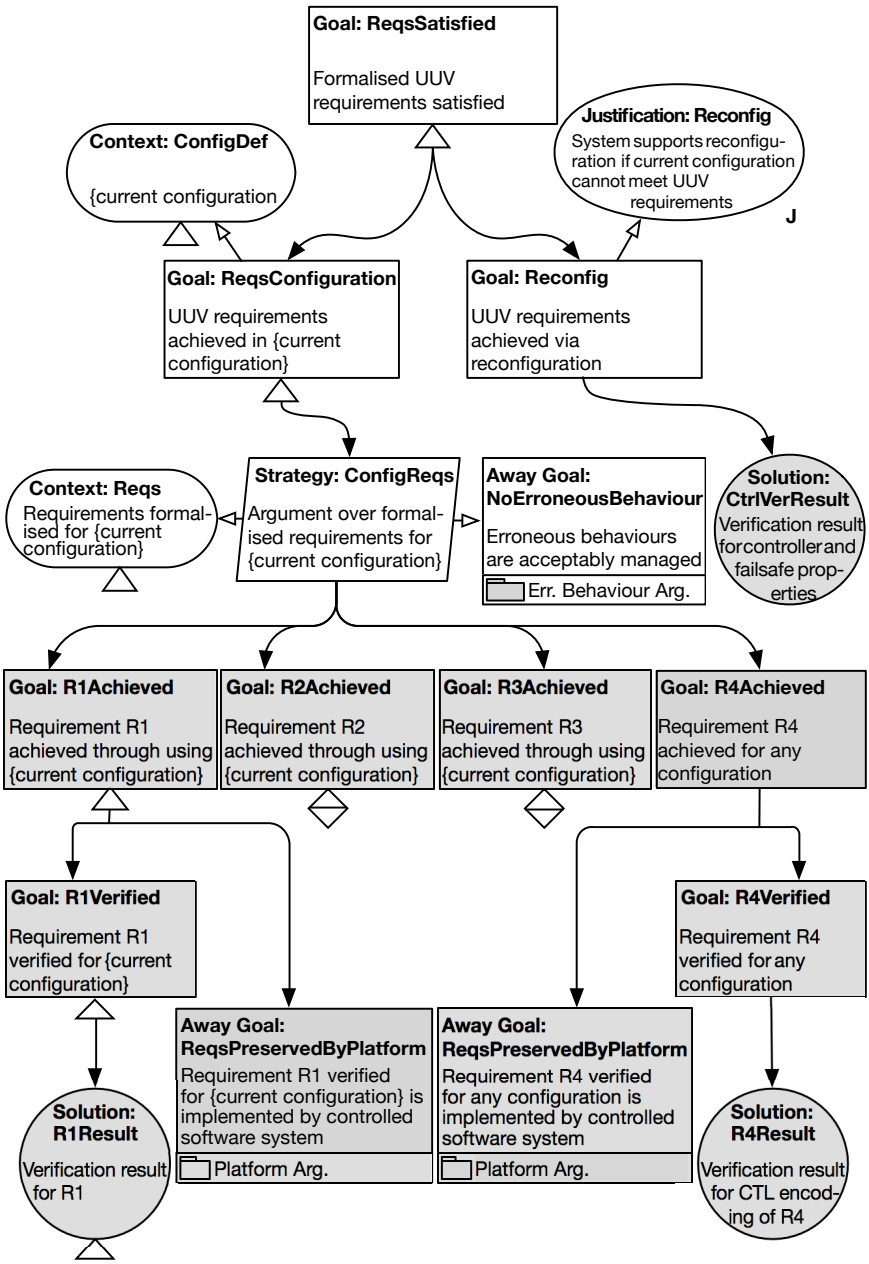


Figure 4.16: Partially-instantiated assurance argument for the UUV system

4.6.4.2 FX System

To assemble the ENTRUST controller for the FX system, we combined the controller and stochastic models from Stage 1 with our generic controller platform,

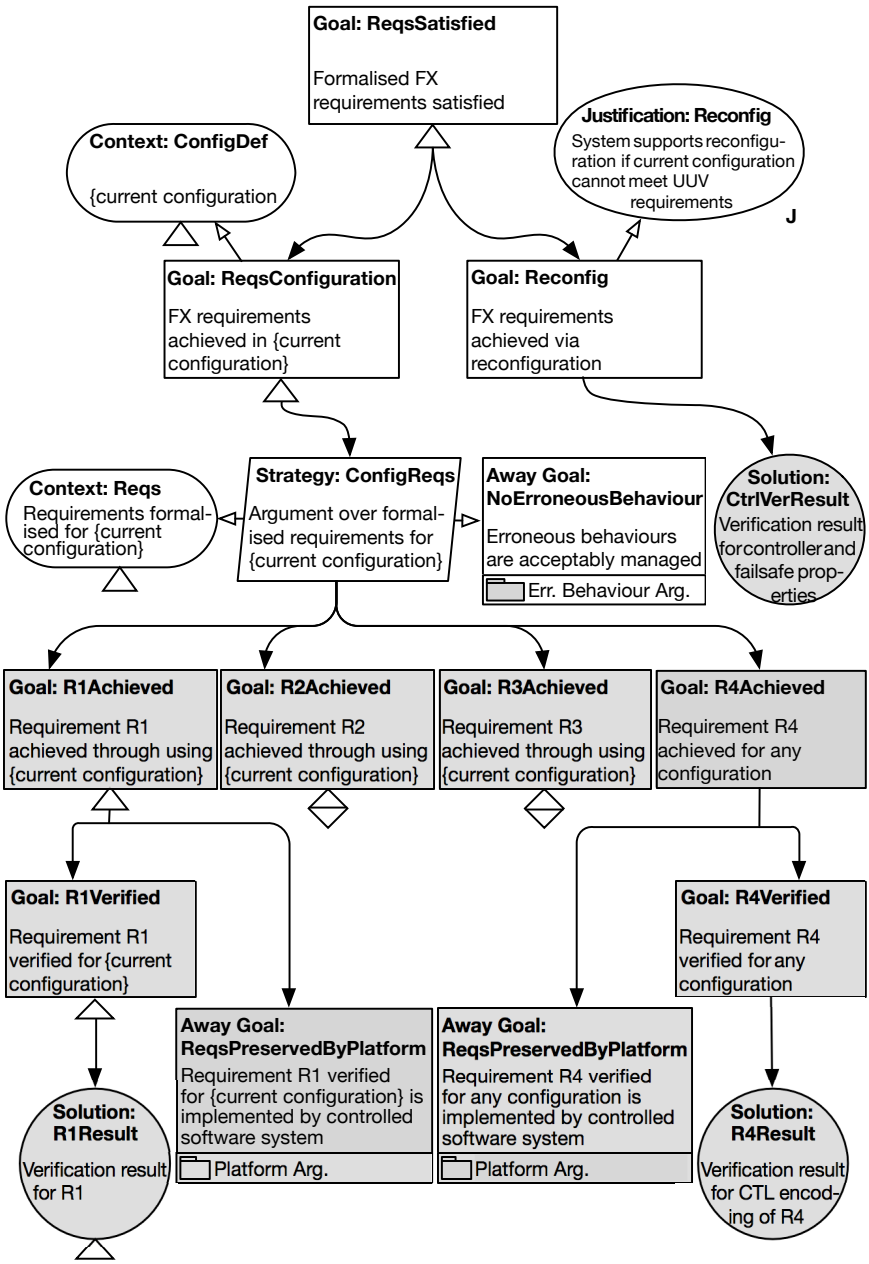


Figure 4.17: Partially-instantiated assurance argument for the FX system; the elements (partially) instantiated in Stage 3 of ENTRUST are shaded.

and with FX-specific Java classes that we implemented to specialise the abstract Sensors, Effectors and VerificationEngine abstract classes of ENTRUST. The

Sensors class synchronises with the Monitor automaton from Fig. 4.14 through the **newServicesCharacteristics!** signal (issued after changes in the properties of the FX services are detected). In addition, the Sensors and Effectors classes use the relevant API methods of an FX implementation that we developed as explained in Section 4.6.5.2. The specialised VerificationEngine instantiates the parametric DTMC model from Fig. 4.15 at runtime, and verifies the PCTL formulae devised for requirements R1–R3 from Section 4.6.1.2.

4.6.4.3 Discussion

The controller enactment comprises typical software development (i.e. specialisation of Java classes) and integration tasks. A considerable part of the required functionality is application-independent, and already provided by the reusable abstract Java classes available with ENTRUST.

4.6.5 Deployment of the Self-Adaptive System

4.6.5.1 UUV System

We used the open-source MOOS-IvP[1] platform (`oceanai.mit.edu/moos-ivp`) for the implementation of autonomous applications on unmanned marine vehicles [20] to develop a fully-fledged three-sensor UUV simulator that is available on the ENTRUST website. We then exploited the publish-subscribe architecture of MOOS-IvP to interface the ENTRUST sensors and effectors (and thus the controller from Section 4.6.4.1) with the UUV simulator, we installed the controller and the controlled system on a computer with a similar spec to that of the payload computer of a mid-range UUV, and we preconfigured the system to start with zero speed and all its sensors switched off. We chose this configuration, corresponding to initial UUV parameter values $(x_1, x_2, x_3, sp) = (0, 0, 0, 0)$, to ensure that the system started with a configuration satisfying its failsafe requirement R4 (cf. Section 4.3.1).[2]

4.6.5.2 FX System

We implemented a prototype version of the FX system using Java web services deployed in Tomcat/Axis, and a Java FX workflow that we integrated with the ENTRUST controller from Stage 4. Our self-adaptive FX system (whose code is available on our project website) could select from two functionally equivalent web service implementations for each of the six FX services from Fig. 4.6, i.e. from 12 web services with the initial characteristics shown in Table 4.5. For simplicity and without loss of generality, we installed the components of the self-adaptive FX system on a single computer with the characteristics detailed in Section 4.7.1, and we preconfigured the system to start by using the first web service implementation available for each service (i.e. MW_0, TA_0, etc.), except for the *Order* service. For *Order*, NoSvc was selected initially, to ensure that the failsafe

[1] Mission-Oriented Operating Suite – Interval Programming

[2] The use of a failsafe initial configuration is our recommended approach for ENTRUST self-adaptive systems. When this is not possible, an execution of the MAPE loop must be initiated as part of the system start-up, to ensure that an initial configuration meeting the system requirements is selected.

Table 4.5: Initial characteristics of the service instances used by the FX system

Operation:	*Market Watch*		*Technical Analysis*		*Fundam. Analysis*		*Alarm*		*Order*		*Notification*	
Service ID:	MW_0	MW_1	TA_0	TA_1	FA_0	FA_1	Al_0	Al_1	Or_0	Or_1	No_0	No_1
response time [s]	.5	.5	.6	1.0	1.6	.7	.6	.9	.6	1.3	1.8	.5
reliability	.976	.995	.998	.985	.998	.99	.995	.99	.995	.95	.99	.99
price	5	10	6	4	23	25	15	9	25	20	5	8

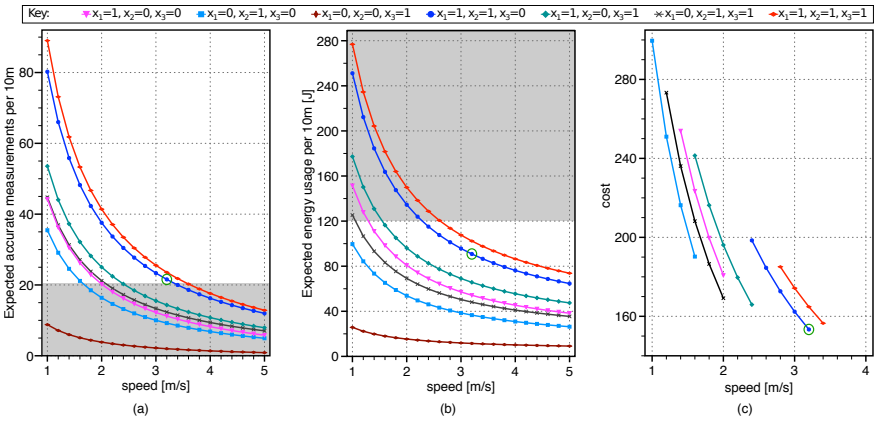


Figure 4.18: Verification results for requirement (a) R1, (b) R2, and (c) cost of the feasible configurations; 21 speed values between 1m/s and 5m/s are considered for each of the seven combinations of active sensors, corresponding to $21 \times 7 = 147$ alternative configurations. The best configuration (circled) corresponds to $x_1 = x_2 = 1$, $x_3 = 0$ (i.e. UUV using only its first two sensors) and $sp = 3.2$m/s, and the shaded regions correspond to requirement violations.

requirement R4 was satisfied until a configuration meeting requirements R1–R3 was automatically selected by the first execution of the MAPE loop, shortly after the system started.

4.6.5.3 Discussion

This stage involved a typical deployment of the managed systems and of their controllers, except that both self-adaptive systems were preconfigured to start with a configuration satisfying their failsafe requirement. Note that such a configuration always exists because the compliance of the two systems with their failsafe requirements was formally verified in the second ENTRUST stage (cf. Section 4.6.2).

4.6.6 Self-Adaptation

4.6.6.1 UUV System

The dynamic reconfiguration of the self-adaptive UUV system is described in detail in Section 4.7.1.1. Here we illustrate the process by considering a scenario in which the UUV system comprises $n = 3$ sensors with: initial measurement rates $r_1 = 5\text{s}^{-1}, r_2 = 4\text{s}^{-1}, r_3 = 4\text{s}^{-1}$; energy consumed per measurement $e_1 = 3$J, $e_2 = 2.4$J, $e_3 = 2.1$J; and energy used for switching a sensor on and off $e_1^{\text{on}} = 10$J, $e_2^{\text{on}} = 8$J, $e_3^{on} = 5$J and $e_1^{\text{off}} = 2$J, $e_2^{\text{off}} = 1.5$J, $e_3^{\text{off}} = 1$J, respectively. Also, suppose that the current UUV configuration is $(x_1, x_2, x_3, sp) = (0, 1, 1, 2.8)$, and that sensor 3 experiences a degradation such that $r_3^{\text{new}} = 1\text{s}^{-1}$. The ENTRUST controller gets this new measurement rate through the monitor. As the sensor rates differ from those in the knowledge repository, the guard *analysisRequired()* returns true and the **startAnalysis!** signal is sent. Upon receiving the signal, the *analyser*

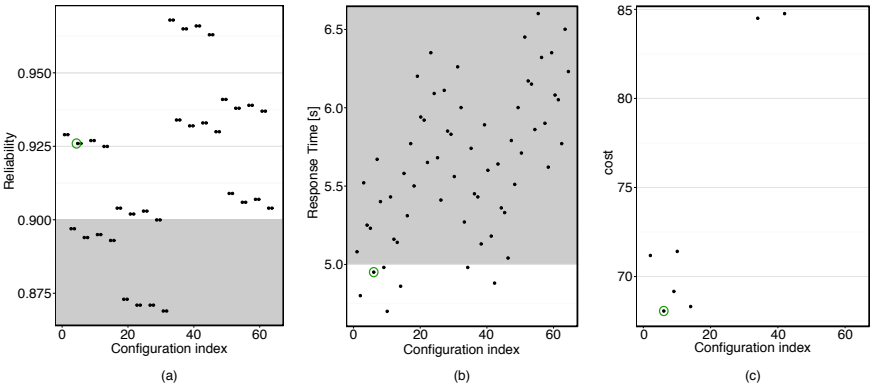


Figure 4.19: Runtime verification results for FX requirement (a) R1, (b) R2, and (c) R3—cost of the feasible configurations, where the configuration index $i_1 i_2 i_3 i_4 i_5 i_6$ in number base 2 corresponds to the FX configuration that uses services MW_{i_1}, TA_{i_2}, FA_{i_3}, AI_{i_4}, Or_{i_5} and No_{i_6}. The best configuration (circled) has index $5_{(10)} = 000101_{(2)}$, corresponding to MW_0, TA_0, FA_0, AI_1, Or_0 and No_1. Shaded regions correspond to requirement violations.

model invokes the probabilistic verification engine, whose analysis results for requirements **R1–R3** are depicted in Fig. 4.18. The *analyse()* action filters the results as follows: configurations that violate requirements **R1** or **R2**, i.e., the shaded areas from Fig. 4.18a and Fig. 4.18b, respectively, are discarded.[3] The remaining configurations are feasible, so their cost (4.1) is computed for $w_1 = 1$ and $w_2 = 200$. The configuration minimising the cost (i.e., $(x_1, x_2, x_3, sp) = (1, 1, 0, 3.2)$ – circled in Fig. 4.18a-c) is selected as the best configuration. Since the best and the current configurations differ, the analyzer invokes the planner to assemble a stepwise reconfiguration plan with which i) sensor 1 is switched on; ii) next, sensor 3 is switched off; and iii) finally the speed is adjusted to 3.2m/s. Once the plan is assembled, the executor is enforcing this plan to the UUV system. The adaptation results from Fig. 4.18 provide the evidence required for the generation of the assurance case as described in Section 4.6.7.1.

4.6.6.2 FX System

In this stage, the self-adaptive FX system dynamically reconfigures in response to observed changes in the characteristics of the web services it uses. Several such reconfigurations are described later in the paper, in Section 4.7.1.2 and in Fig. 4.23. To illustrate this process in detail, consider the system configuration

[3] Note that R1 and R2 are "conflicting" requirements, in the sense that the configurations that satisfy R1 by the widest margin violate R2, and the other way around. In such scenarios, ENTRUST supports the selection of configurations based on trade-offs between the conflicting requirements, as specified by a cost (or utility) function. If either requirement became much stricter (e.g. if R1 required over 50 measurements per every 10m), no configuration would satisfy both R1 and R2. In this case, ENTRUST would choose the configuration specified by the failsafe requirement R4, i.e. would reduce the UUV speed to 0m/s, and would record the probabilistic model checking evidence showing the lack of a suitable non-failsafe configuration.

immediately after change C from Fig. 4.23, where the FX workflow uses the services MW_1, TA_0, FA_0, AI_0, Or_0 and No_1. This configuration is reached after the FX services, initially operating with the characteristics from Table 4.5, experience degradations in the reliability of MW_0 ($p^{\mathsf{new}}_{\mathsf{MW}_0} = 0.9$, change B in Fig. 4.23) and in the response time of FA_1 ($time^{\mathsf{new}}_{\mathsf{FA}_1} = 1.2$s, change C in Fig. 4.23). With the FX system in this configuration, suppose that the *Market Watch* service MW_0 recovers, i.e., $p^{\mathsf{new}}_{\mathsf{MW}_0} = 0.976$ as in Table 4.5. Under these circumstances, which correspond to change D from Fig. 4.23, the ENTRUST controller receives the updated characteristics of MW_0 via its monitor. As the new service characteristics differ from those in the knowledge repository, the guard *analysisRequired()* holds and the **startAnalysis!** signal is sent. The analyser model receives the signal and invokes the runtime probabilistic verification engine, whose analysis of the FX requirements R1–R3 over the $2^6 = 64$ possible system configurations (corresponding to six services each provided by two implementations) is shown in Fig. 4.19. As part of this analysis, configurations that violate requirements R1 or R2 (i.e., those from the shaded areas in Fig. 4.19a and Fig. 4.19b, respectively) are discarded. The remaining configurations are feasible, so their cost is calculated (for $w_1 = 1$ and $w_2 = 2$) as shown in Fig. 4.19c. The feasible configuration using services MW_0, TA_0, FA_0, AI_1, Or_0 and No_1 has the lowest cost and is thus selected as the best system configuration. Since the best and the current configurations differ, the guard *adaptationRequired()* holds and the analyser invokes the planner through the **startPlanning!** signal to assemble a stepwise reconfiguration plan through which: (i) MW_0 replaces MW_1; and (ii) AI_1 replaces AI_0. Once the plan is ready, the executor automaton receives the **startExecuting?** signal and is ensuring the implementation of this plan by sending the signal **changeService!** to the system effectors.

4.6.6.3 Discussion

Both self-adaptive systems reconfigured in response to application-specific changes (more of which are described in Section 4.7). Selecting the new configurations involved the runtime probabilistic model checking of different types of stochastic models, to generate assurance evidence that system requirements were satisfied after each change.

4.6.7 Synthesis of Dynamic Assurance Argument

4.6.7.1 UUV System

In this stage, the partially-instantiated assurance argument pattern for the UUV system (Fig. 4.16) is fully instantiated after every selection of a new UUV configuration by the ENTRUST controller. For instance, after the ENTRUST controller activities described in Section 4.6.6.1 conclude with the selection of the UUV configuration $(x_1, x_2, x_3, sp) = (1, 1, 0, 3.2)$ and the generation of runtime verification evidence that this configuration satisfies requirements R1–R3, this partially-instantiated assurance argument pattern is fully instantiated as shown in Fig. 4.20.

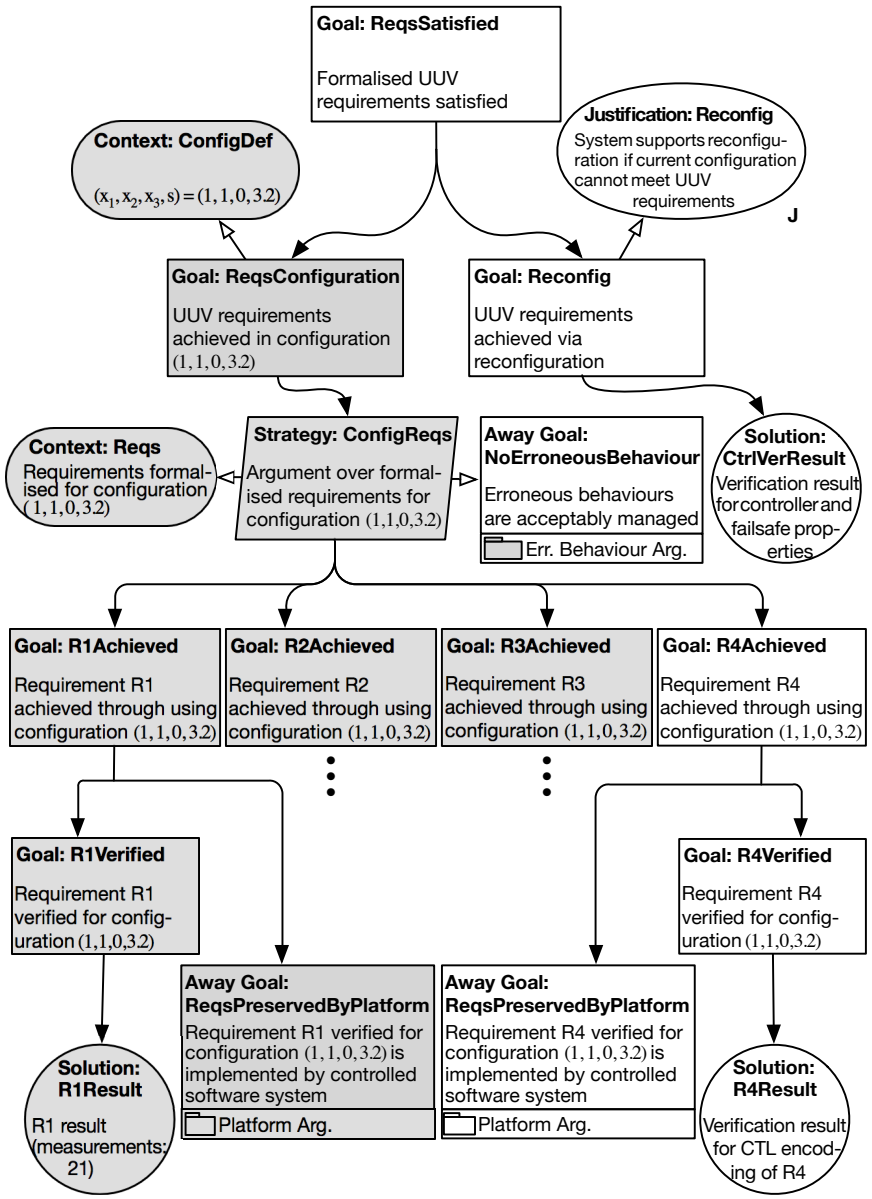


Figure 4.20: Fully-instantiated assurance argument for the UUV system; the subgoals for **R2Achieved** and **R3Achieved** (not included due to space constraints) are similar to those for **R1Achieved**, and shading is used to show the elements instantiated at runtime

4.6.7.2 FX System

The partially instantiated FX assurance pattern from Fig. 4.17 is updated into a full assurance argument after each selection of a new configuration by the EN-

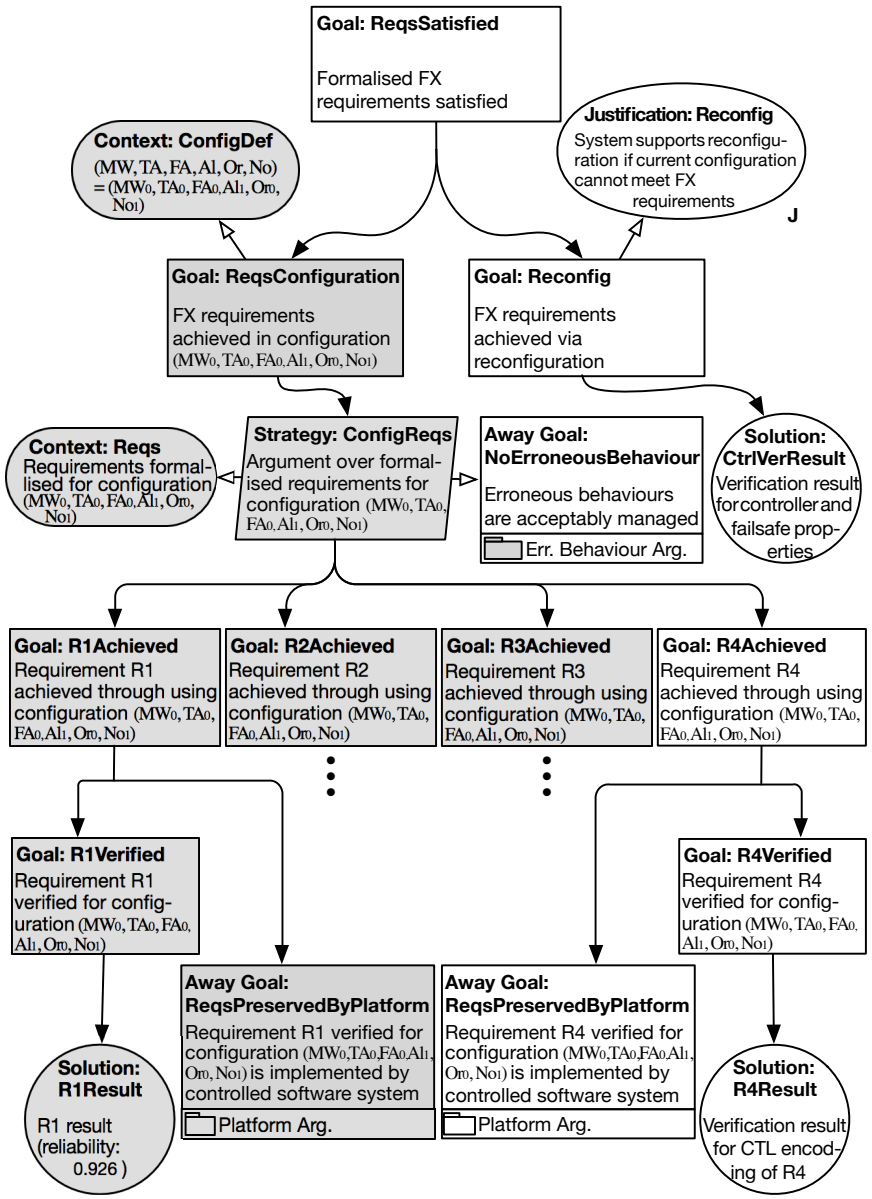


Figure 4.21: Fully-instantiated assurance argument for the FX system; the subgoals for **R2Achieved** and **R3Achieved** (not included due to space constraints) are similar to those for **R1achieved**, and shading is used to show the elements instantiated at runtime

TRUST controller. This involves using the new evidence generated by the runtime probabilistic verification engine to complete the instantiation of the assurance pattern. As an example, Fig. 4.21 shows the complete assurance pattern synthesised

as part of the configuration change that we used to illustrate the previous stage of ENTRUST in Section 4.6.6.2.

4.6.7.3 Discussion

For both the UUV system and the FX system, integrating the dynamically generated assurance evidence required the updating of only a few uninstantiated GSN '*solutions*' from the partially instantiated assurance arguments. In contrast, instantiating the current system configurations in the two assurance arguments involved multiple but small updates of '*context*', '*strategy*' and '*goal*' GSN elements. The rightmost branches of the assurance arguments ensure the goals associated with the failsafe requirements of the two systems, and therefore remained unchanged in this ENTRUST stage.

4.7 Evaluation

To evaluate the effectiveness and generality of ENTRUST, we used our methodology to engineer the self-adaptive software systems from Section 4.3. The two systems were developed as described in Section 4.6, and were deployed in a realistic environment seeded with simulated changes specific to their application domains. Finally, we examined the correctness and efficiency of the adaptation and of the assurance cases produced by ENTRUST in response to each of these unexpected environmental changes. The aim of our evaluation was to answer the following research questions.

RQ1 (Correctness): Are ENTRUST self-adaptive systems making the right adaptation decisions and generating valid assurance cases?

RQ2 (Efficiency): Does ENTRUST provide design-time and runtime assurance evidence with acceptable overheads for realistic system sizes?

RQ3 (Generality): Does ENTRUST support the development of self-adaptive software systems and dynamic assurance cases across application domains?

As the focus of our evaluation was the ENTRUST methodology and its tool-supported instance, we necessarily made a number of assumptions. In particular, we assumed that established assurance processes could be used to construct assurance arguments for all aspects of the controlled systems from our case studies, including their correct design, development, operation, ability to respond to effector requests, and any real-time considerations associated with achieving the new configurations decided by the ENTRUST controller. As such, these aspects are outside the scope of ENTRUST and are not covered in our evaluation. We further assumed that the derivation, validity, completeness and formalisation of the self-adaptive system requirements are addressed as part of the overall system assurance cases for the two case studies, and therefore also outside the scope of our evaluation of ENTRUST.

The experiments carried out to address the three research questions are described in Sections 4.7.1–4.7.3, and the main threats to validity are discussed in Section 4.7.4.

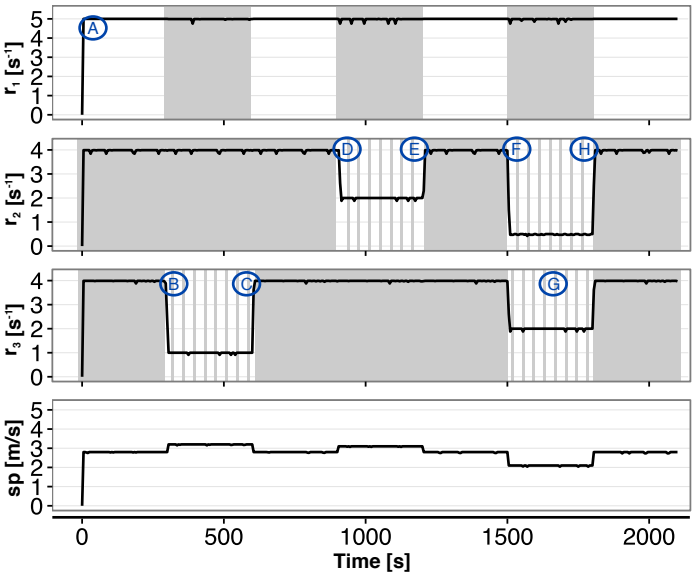


Figure 4.22: Change scenarios for the self-adaptive UUV system over 2100 seconds of simulated time. Extended shaded regions indicate the sensors switched on at each point in time, and narrow shaded areas show the periodical testing of sensors switched off due to degradation (to detect their recovery).

4.7.1 RQ1 (Correctness)

To answer the first research question, we carried out experiments that involved running the UUV and FX systems in realistic environments comprising (simulated) unexpected changes specific to their domains. For the UUV system, the experiments were seeded with failures including sudden degradation in the measurement rates of sensors and complete failures of sensors, and with recoveries from these problems. For the FX system, we considered variations in the response time and the probability of successful completion of third-party service invocation. All the experiments were run on a MacBook Pro with 2.5 GHz Intel Core i7 processor, and 16 GB 1600 MHz DDR3 RAM.

4.7.1.1 UUV System

For the UUV system, we described a concrete change scenario and the resulting self-adaptation process and generation of an assurance case in Sections 4.6.6.1 and 4.6.7.1. The complete set of change scenarios we used in this experiment is summarised in Fig. 4.22, which depicts the changes in the sensor rates and the new UUV configurations selected by the ENTRUST controller. The labels A–H from Fig. 4.22 correspond to following key events:

A) The UUV starts with the initial state and configuration from Section 4.6.6.1;

B) Sensor 3 experiences the degradation described in Section 4.6.6.1 ($r_3^{\text{new}} = 1$), so the higher-rate but less energy efficient sensor 1 is switched on (allowing a slight increase in speed to $sp = 3.2$m/s) and sensor 3 is switched off;

C) Sensor 3 recovers and the initial configuration is resumed;

D) Sensor 2 experiences a degradation, and is replaced by sensor 1, with the speed increased to $sp = 3.1$m/s;

E) Sensor 2 recovers and the initial configuration is resumed;

F) Both sensor 2 and sensor 3 experience degradations, so sensor 1 alone is used, with the UUV travelling at a lower speed $sp = 2.1$m/s;

G) Periodic tests (which involve switching sensors 2 and 3 on for short periods of time) are carried out to detect a potential recovery of the degraded sensors;

H) Sensors 2 and 3 resume operation at nominal rates and the initial UUV configuration is reinstated.

If the UUV system was not self-adaptive, it would have to operate with a fixed configuration, which would lead to requirement violations for extended periods of time. To understand this drawback of a non-adaptive UUV, consider that its fixed configuration is chosen to coincide with the initial UUV configuration from Fig. 4.22 (i.e. $(x_1, x_2, x_3, sp) = (0, 1, 1, 2.8)$) – a natural choice because manual analysis can be used to find that this configuration satisfies the UUV requirements at deployment time. However, with this fixed configuration, the UUV will violate its throughput requirement R1 whenever one or both of UUV sensors 1 and 2 experience a non-trivial degradation, i.e. in the time intervals B–C (only 13 measurements per 10m instead of the required 20 measurements, according to additional analysis we carried out), D–E (only 15 measurements per 10m) and F–H (only 7 measurements per 10m) from Fig. 4.22. Although a different fixed configuration may always meet requirement R1, such a configuration would violate other requirement(s), e.g. having all three UUV sensors switched on meets R1 but violates the resource usage requirement R2 at all times.

Finally, we performed experiments to assess how the adaptation decisions may be affected by changes in the weights w_1, w_2 from the UUV cost (4.1) and the energy usage of the n UUV sensors. We considered UUVs with $n \in \{3, 4, 5, 6\}$ sensors, and for each value of n we carried out 30 independent experiments with the weights w_1, w_2 randomly drawn from the interval $[1, 500]$, and the energy consumption for taking a measurement and switching on and off a sensor (i.e., e_i, e_i^{on} and e_i^{off}, $1 \leq i \leq n$) randomly drawn from the interval $[0.1J, 10J]$. The experimental results (available, together with the PRISM-generated assurance evidence, on the project website) show that ENTRUST successfully reconfigured the system irrespective of the weight and energy usage values. In particular, if a configuration satisfying requirements R1–R3 existed for a specific change and system characteristics combination, ENTRUST reconfigured the UUV system to use this configuration. As expected, the configuration minimising the cost (4.1) depended both on the values of the weights w_1, w_2 and on the sensor energy usage. When no configuration satisfying requirements R1–R3 was available, ENTRUST employed the zero-speed failsafe configuration from requirement R4 until configurations satisfying requirements R1–R3 were again possible after a sensor recovery.

4.7.1.2 FX System

For the FX system, a concrete change scenario is detailed in Section 4.3.2, and the complete set of change scenarios used in our experiments is summarised in Fig. 4.23, where labels A–G correspond to the following events:

A) The FX starts with the initial services characteristics from Table 4.5 and uses a configuration comprising the services MW_0, TA_0, FA_0, AI_1, Or_0 and No_1, which satisfies requirements R1 and R2 and optimises R3;

B) The *Market Watch* service MW_0 experiences a significant reliability degradation ($p^{\mathsf{new}}_{\mathsf{MW}_0} = 0.9$), so FX starts using the significantly more reliable MW_1, and thus "affords" to also switch to the slightly less reliable but faster *Fundamental Analysis* service FA_1 in order to minimise the *cost* defined in requirement R3;

C) Due to an increase in response time of *Fundamental Analysis* service FA_1 ($time^{\mathsf{new}}_{\mathsf{FA}_1} = 1.2\text{s}$), the FX switches to using FA_0 and also replaces the *Alarm* service AI_1 with the faster but more expensive service AI_0 (to meet the timing requirement R2);

D) The *Market Watch* service MW_0 recovers, so FX switches back to this services and also resumes using the less reliable *Alarm* service AI_1;

E) The *Technical Analysis* service TA_0 and the *Notification* service No_1 exhibit unexpected degradations in reliability ($p^{\mathsf{new}}_{\mathsf{TA}_0} = 0.98$) and in response time ($time^{\mathsf{new}}_{\mathsf{No}_1} = 1\text{s}$), respectively, so the FX system self reconfigures to use MW_0, TA_1, FA_1, AI_0, Or_0 and No_0;

F) As a result of a reliability degradation in the *Order* service Or_0 ($p^{\mathsf{new}}_{\mathsf{Or}_0} = 0.91$) and recovery of the *Technical Analysis* service TA_0, the FX system replaces services MW_0, TA_1, FA_1 and Or_0 with MW_1, TA_0, FA_0 and Or_1, respectively;

G) All the degraded services recover, so the initial configuration MW_0, TA_0, FA_0, AI_1, Or_0 and No_1 is reinstated.

As in the case of the UUV system, a non-adaptive FX version will fail to meet the system requirements for extended periods of time. For example, choosing to always use the initial FX configuration from Fig. 4.23 would lead to a violation of the reliability requirement R1 while service MW_0 experiences a significant reliability degradation in the time interval B–D. While using service MW_1 instead of MW_0 would avoid this violation, MW_1 is more expensive but no faster than MW_0 (cf. Table 4.5) so its choice would increase the cost (4.2), thus violating the cost requirement R3 in the time interval A–B.

4.7.1.3 Discussion

For each change scenario from our experiments within the two case studies (cf. Figs. 4.22 and 4.23), we performed two checks. For the former check, we confirmed that the ENTRUST controller operated correctly. To this end, we established that the change was accurately reported by the sensors and correctly processed by the monitor, leading the analyzer to select the right new configuration, for which a correct plan was built by the planner and implemented by the executor.

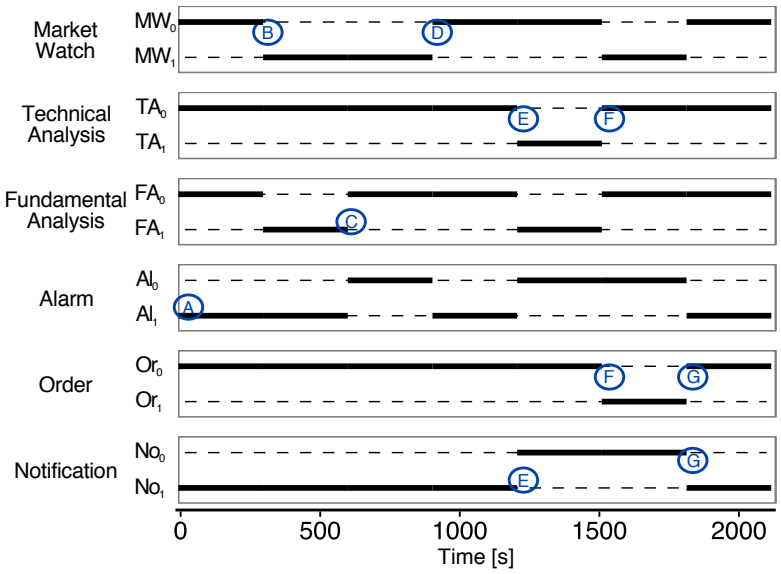


Figure 4.23: Change scenarios for the self-adaptive FX system, with the initial services characteristics shown in Table 4.5. The thick continuous lines depict the services selected at each point in time.

For the latter check, we determined the suitability of the ENTRUST assurance cases. We started from the guidelines set by safety and assurance standards, which highlight the importance of demonstrating, using available evidence, that an assurance argument is *compelling*, *structured* and *valid* [56, 137, 193]. Also, we considered the fact that ENTRUST has been examined experimentally but has not been tested in real-world scenarios to generate the industrial evidence necessary before approaching the relevant regulator. However, our preliminary results show, based on formal design-time and runtime evidence, that the primary claim of ENTRUST assurance cases is supported by a direct and robust argument. Firstly, the argument assures the achievement of the requirements either based on a particular active configuration or through reconfiguration, while maintaining a failsafe mechanism. Secondly, the argument and patterns are well-structured and conform to the GSN community standard [95]. Thirdly, ENTRUST provides rigorous assessments of validity not only at design time but also through-life, by means of monitoring and continuous verification that assess and challenge the validity of the assurance case based on actual operational data. This continuous assessment of validity is a core requirement for safety standards, as highlighted recently for medical devices [162]. As such, our approach satisfies five key principles of dynamic assurance cases [61]:

- *continuity* and *updatability*, as evidence is generated and updated at runtime to ensure the continuous validity of the assurance argument (e.g. the formal evidence for solution **R1Result** from the UUV argument in Fig. 4.20, which satisfies a system requirement given the current configuration);

- *proactivity*, since the assurance factors that provide the basis for the evidence in the assurance argument are proactively identified (e.g. the **ConfigDef** context from the UUV argument in Fig. 4.20, which captures the parameters of the current configuration);
- *automation*, because the runtime evidence is dynamically synthesised by the MAPE controller;
- *formality*, as the assurance arguments are formalised using the GSN standard.

In conclusion, subject to the limitations described above, our experiments provide strong empirical evidence that ENTRUST self-adaptive systems make the right adaptation decisions and generate valid assurance cases.

4.7.2 RQ2 (Efficiency)

To assess the efficiency of the ENTRUST generation of assurance evidence, we measured the CPU time taken by (i) the design-time UPPAAL model checking of the generic controller properties from Table 4.4; and (ii) the runtime probabilistic model checking performed by the ENTRUST analyzer. Fig. 4.24 shows the time taken to verify the generic controller properties from Table 4.4 for a three-sensor UUV system, and for an FX system comprising two third-party implementations for each workflow service. With typical CPU times of several minutes per property and a maximum below 12 minutes, the overheads for this design-time, once-only verification of all controller properties are entirely acceptable.

The CPU times required for the runtime probabilistic model checking of the QoS requirements for alternative configurations of the two systems (Fig. 4.25) have values below 1.5s and 2s, respectively. These runtime overheads, which correspond to under 10ms for the verification of a UUV configuration and under 30ms for the verification of an FX configuration, are acceptable because ENTRUST is intended for scenarios where:

1) failures and other changes requiring system reconfigurations are, on average, much less frequent than the frequency with which the runtime verification can be executed (i.e. every 1.5–2s for our two systems);
2) failsafe configurations can be temporarily assumed if needed during the infrequent reverifications of the ENTRUST stochastic models.

These assumptions ensure that, most of the time, ENTRUST adaptation decisions are reached before new changes occur and can be applied. They also ensure that any time spent in failsafe configurations is small compared to the time when the system employs "useful" configurations.

As shown in Fig. 4.25, we also ran experiments to assess the increase in runtime overhead with the system size and number of alternative configurations, by considering UUVs with up to six sensors, and FX system variants with up to five implementations per service. Typical for model checking, the CPU time increases exponentially with these system characteristics. This makes the current implementation of our ENTRUST instance suitable for self-adaptive systems with up to hundreds of configurations to analyse and select from at runtime. However, our recent work on compositional [38], incremental [113], caching-lookahead [87] and

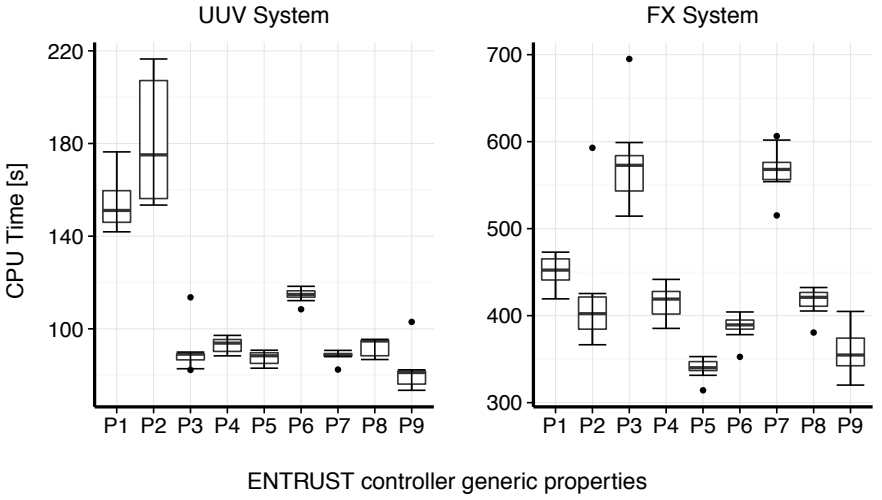


Figure 4.24: CPU time for the UPPAAL verification of the generic controller properties in Table 4.4 (box plots of 10 independent measurements)

distributed [34] approaches to probabilistic model checking and on metaheuristics for probabilistic model synthesis [88] suggests that these more efficient model checking approaches could be used to extend the applicability of our ENTRUST instance to much larger configuration space sizes. As an example, in [87] we used *caching* of recent runtime probabilistic model checking results and anticipatory verification of likely future configurations (i.e. *lookahead*) to significantly reduce the mean time required to select new configurations for a variant of our self-adaptive UUV system (by over one order of magnitude in many scenarios). Integrating ENTRUST with these approaches is complementary to the purpose of this paper and represents future work.

4.7.3 RQ3 (Generality)

We used ENTRUST to develop an embedded system from the oceanic monitoring domain, and a service-based system from the exchange trade domain. As previously mentioned in Section 4.3 and summarised in Table 4.1, self-adaptation within these systems was underpinned by the verification of continuous- and discrete-time Markov chains, respectively; and the requirements and types of changes for the two systems differed. Finally, the ENTRUST assurance arguments for the two systems were based on assurance evidence obtained using multiple verification techniques:

1) testing evidence for the correct operation of trusted virtual machine;
2) model checking evidence for the correctness of the MAPE controller and the failsafe system requirements;
3) probabilistic model checking evidence for the remaining system requirements.

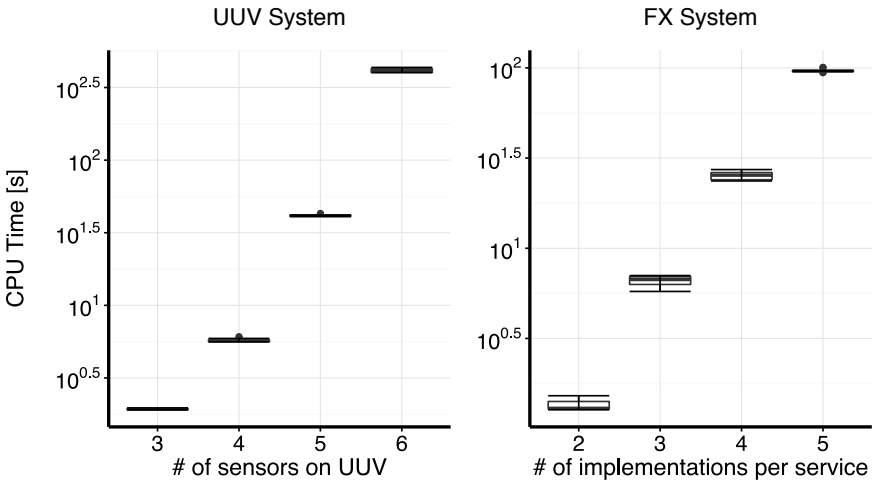


Figure 4.25: CPU time for the runtime probabilistic model checking of the QoS requirements after changes (box plots based on 10 system runs comprising seven changes each—70 measurements in total)

Although evaluation in additional areas is needed, these results indicate that our ENTRUST instance can be used across application domains.

To assess the overall generality of ENTRUST, we note that probabilistic model checking can effortlessly be replaced with simulation in our experiments, because the probabilistic model checker PRISM can be configured to use discrete-event simulation instead of model checking techniques. Using this PRISM configuration requires no change to the Markov models or probabilistic temporal logic properties we analysed at runtime. As for any simulation, the analysis results would be approximate, but would be obtained with lower overheads than those from Fig. 4.25.

The uncertainties that affect self-adaptive systems are often of a stochastic nature, and thus the use of stochastic models and probabilistic model checking to analyse the behaviour of these systems is very common (e.g. [37, 38, 41, 68, 71, 76, 158, 180]). As such, our ENTRUST instance is applicable to a broad class of self-adaptive systems.

Nevertheless, other methods have been used to synthesise MAPE controllers and to support their operation. Many such methods (e.g. based on formal proof, traditional model checking, other simulation techniques and testing) are described in Section 4.8. Given the generality of ENTRUST, these methods could potentially be employed at design time and/or at runtime by alternative instantiations of ENTRUST, supported by different modelling paradigms, requirement specification formalisms, and tools. For example, the use of the (non-probabilistic) graph transformation models or dynamic tests proposed in [15] and [78], respectively, in the self-adaptation ENTRUST stage is not precluded by any of our assumptions (cf. Section 4.4.2.1), although the method chosen for this stage will clearly constrain the types of requirements for which assurance evidence can be provided at runtime.

4.7.4 Threats to Validity

Construct validity threats may be due to the assumptions made when implementing our simple versions of the UUV and FX systems, and in the development of the stochastic models and requirements for these systems. To mitigate these threats, we implemented the two systems using the well-established UUV software platform MOOS-IvP and (for FX) standard Java web services deployed in Tomcat/Axis. The model and requirements for the UUV system are based on a validated case study that we are familiar with from previous work [87], and those for the FX system were developed in close collaboration with a foreign exchange expert.

Internal validity threats can originate from how the experiments were performed, and from bias in the interpretation of the results due to researcher subjectivity. To address these threats, we reported results over multiple independent runs; we worked with a team comprising experts in all the key areas of ENTRUST (self-adaptation, formal verification and assurance cases); and we made all experimental data and results publicly available to enable replication.

External validity threats may be due to the use of only two systems in our evaluation, and to the experimental evaluation having been done by only the authors' three research groups. To reduce the first threat, we selected systems from different domains with different requirements. The evaluation results show that ENTRUST supports the development of trustworthy self-adaptive solutions with assurance cases for the two different settings. To reduce the second threat, we based ENTRUST on input from, and needs identified by, the research community [7, 49, 134, 135]. In addition, we fine tuned ENTRUST based on feedback from industrial partners involved in the development of mission-critical self-adaptive systems, and these partners are now using our methodology in planning future engineering activities. Nevertheless, additional evaluation is required to confirm generality for domains with characteristics that differ from those in our evaluation (e.g., different timing patterns and types of requirements and disturbances) and usability by a larger number of users.

4.8 Related Work

Given the uncertain operating conditions of self-adaptive systems, a central aspect of providing assurances for such systems is to collect and integrate evidence that the requirements are satisfied during the entire lifetime. To this end, researchers from the area of self-adaptive systems have actively studied a wide variety of assurance methods and techniques applicable at design time and/or at runtime[49, 135, 140, 186, 207, 213, 224]. Tables 4.6 and 4.7 summarise the state of the art, partitioned into categories based on the main method used to provide assurances, e.g. formal proof, model checking or simulation. We consider as the main method of a study from our analysis the method that the study primarily focuses on; the approaches from these studies may implicitly use additional methods, such as testing of their platforms and tools, but this is not emphasised by their authors. We

summarise the representative approaches included in each category according to their:

1) *Assurances evidence*, comprising separate parts for the methods used to provide assurance evidence for: (i) the correctness of the platform used to execute the controller, (ii) the correctness of the controller functions, and (iii) the correctness of the runtime adaptation decisions;
2) *Methodology*, comprising three parts: the engineering process (i.e. a methodical series of steps to provide the assurances), tool support (i.e., tools used by engineers to provide evidence at design time and tools used at runtime by the controller, e.g. during analysis or planning), and other reusable components (i.e. third-party libraries and purpose-built software components used as part of the controller, and other artefacts that can be used at design time or at runtime, including models, templates, patterns, algorithms).

Providing assurances for self-adaptive systems with strict requirements requires covering all these aspects, as well as an *assurance argument* that integrates the assurance evidence into a compelling, comprehensible and valid case that the system requirements are satisfied. Unlike ENTRUST (Table 4.8), the current research disregards this need for an assurance argument. We discuss below the different approaches and point out limitations that we overcome with ENTRUST.

Formal proof establishes theorems to prove properties of the controller or the system under adaptation. Proof was used to provide evidence for safety and liveness properties of self-adaptive systems with different semantics (one-point adaptation, overlap adaptation, and guided adaptation) [221]. Formal proof was also used to provide evidence for properties of automatically synthesised controllers, e.g. the completeness and soundness of synthesised behavioral models that satisfy an expressive subset of liveness properties [62] and correctness and deadlock free adaptations performed by automatically synthesised controllers [118]. Finally, formal proof was used to demonstrate the correctness of adaptation effects, e.g. proofs for safety, no deadlock, and no starvation of system processes as a result of adaptation [27], and guarantees for the required qualities of adaptations, e.g. proofs for optimised resource allocation, while satisfying quality of service constraints [3]. The focus of all these approaches is on providing assurance evidence for particular aspects of adaptation. All of them offer reusable components, however, these solutions require complete specifications of the system and its environment, and—unlike ENTRUST—cannot handle aspects of the managed system and its environment that are unknown until runtime.

Model checking enables verifying that a property holds for all reachable states of a system, either offline by engineers and/or online by the controller software. Model checking was used to ensure correctness of the adaptation functions that are modeled as interacting automata, with the verified models directly interpreted during execution by a thoroughly tested virtual machine [109]. Model checking was also used to provide guarantees for automatic controller synthesis and enactment, e.g. to assure that a synthesised controller and reusable model interpreter have no anomalies [26]. Model checking has extensively been used to provide guarantees for the effects of adaptation actions on the managed system, e.g. for

Table 4.6: Overview of related research on assurances for self-adaptive systems - part I

Approach	Assurance evidence			Methodology		
	Controller platform	**Controller functions**	**Adaptation decisions**	**Engineering process**	**Tool support**	**Other reusable components**
Formal proof						
Adaptation semantics [221]		Proof of safety and liveness properties of adaptive programs and program compositions				Model checking algorithm
Synthesis of behavioral models [62]		Proof of completeness and soundness of synthesized behavioral models				Controller synthesis technique
Controller synthesis [118]			Proof that controller synthesis algorithm generates controllers that guarantee correct and deadlock free adaptations	Controller synthesis process only	Tool to generate controller offline	Controller synthesis algorithm
Correctness adaptation effects [27]			Proof of safety, no deadlock, and no starvation of system processes as a result of adaptation			Verified middle-ware that ensures safety and liveness of monitored system
Guaranteed qualities [3]			Proof of optimizing resource allocation under QoS constraints			Ad-hoc solver of op-timisation problem
Model checking						
Correct adaptation functions [109]	Thoroughly tested virtual machine used to interpret and run controller models	UPPAAL model checking of interacting timed automata to ensure controller deadlock freeness, liveness, etc. and functional system requirements		UPPAAL used to verify controller models at design time		Tested reusable virtual machine; controller model templates
Controller synthesis and enactment [26]		Synthesised controller that is guaranteed not to be anomalous			Tool used for controller synthesis	Reusable inter-preter and config-uration manager for controller enactment
Safe adaptation configurations [15]			Verification of safety properties of system transitions using a graph transformation model			Symbolic verification procedure
Guaranteed qualities [33]			Probabilistic model checking of continually updated stochastic models of the controlled system and the environment to ensure non-functional requirements			PRISM verification library for analysis of stochastic system and envir- onment models
Resilience to controller failures [43]			Probabilistic model checking of resilience properties of synthesized Markov models of the managed system	Procedure to check resilience to controller failures		Reusable operational profiles to check resilience

Table 4.7: Overview of related research on assurances for self-adaptive systems - part II

Approach	Assurance evidence			Methodology		
	Controller platform	**Controller functions**	**Adaptation decisions**	**Engineering process**	**Tool support**	**Other reusable components**
Simulation						
Evaluation novel approach [182]			Offline simulation to ensure the scalability and robustness to node failures and message loss			
Support for design [41]			Offline simulations to check if the performance of a latency-aware adaptation algorithm falls within predicted bounds		OMNeT++ simulator for checking algorithm performance	
Runtime analysis [209]			Runtime simulation of stochastic models of managed system and environment to ensure non-functional requirements with certain level of confidence		UPPAAL-SMC used for online simulation of stochastic system and environment models	
Testing						
Test effectiveness of adaptation framework [84]			Offline stress testing in client-server system, showing that self-repair significantly improves system performance			Rainbow framework to realise self-adaptation
Test controller robustness [43]		Robustness testing of controller by injecting invalid inputs at the controller's interface and employ responses to classify robustness		Robustness testing procedure only		Probabilistic response specification patterns for robustness testing
Runtime testing [78]			Dynamic tests to validate safe and correct adaptation of system using test cases adapted to changes in the system and environment	One-stage process for test case adaptation		
Other approaches						
Control-theoretic approaches, e.g., [75]		Control-theoretic guarantees for one goal (setpoint) using automatically synthesised controller at runtime	Controller guarantees for stability, overshoot, setting time and robustness of system operating under disturbances		ARPE tool to build online a first-order model of the system	Kalman filter and change point detection procedure for model updates
Runtime verification [179]			Online verification of the probability that a temporal property is satisfied given a sample execution trace		TRACE-CONTRACT tool used for trace analysis	
Sanity checks [197]			Sanity checks evaluate the correctness of resource sharing decisions made by a reasoning engine		CHAMELEON tool providing performance guarantees	

safety properties of the transitions of a managed system that is modeled as a graph transformation system [15], to ensure non-functional requirements by runtime verification of continually updated stochastic models of the controlled system and the environment [33], and to provide evidence for resilience properties of synthesized Markov models of the managed system [43]. Again, the focus of all the approaches is on providing assurance evidence for particular aspects of adaptation. The ENTRUST instance presented in Section 4.5 uses two of these techniques (i.e., [109] and [33]) to verify the correctness of the MAPE logic at design time and to obtain evidence that adaptation decisions are correct at runtime, respectively. In addition, ENTRUST offers a process for the systematic engineering of all components of the self-adaptive system, which includes employing an industry-adopted standard for the formalization of assurance arguments.

Simulation approaches provide evidence by analysing the output of the execution of a model of the system. Simulation was used to evaluate novel self-adaptation approaches, e.g. to ensure the scalability and robustness to node failures and message loss of a self-assembly algorithm [182], and to support the design of self-adaptive systems, e.g., to check if the performance of a latency-aware adaptation algorithm falls within predicted bounds [41]. Recently some efforts have been made to let the controller exploit simulation at runtime to support analysis, e.g. runtime simulation of stochastic models of managed system and environment has been used to ensure non-functional requirements with certain level of confidence [209]. The primary focus of simulation approaches has been on providing assurance evidence for the adaptation actions (either as a means to check the controller effects or to make a prediction of the expected effects of different adaptation options). The approaches typically rely on established simulators.

Testing is a standard method for assessing if a software system performs as expected in a finite number of scenarios. Testing was used to test the effectiveness of adaptation frameworks, e.g. checking whether a self-repair framework applied to a client-server system keeps the latencies of clients within certain bounds when the network is overloaded [84]. Testing was used to provide evidence for the robustness of controllers by injecting invalid inputs at the controller's interface and use the responses to classify robustness [43]. Several studies have applied testing at runtime, e.g. to validate safe and correct adaptations of the managed system based on adapt test cases generated in response to changes in the system and environment [78]. While simulation and testing approaches can be employed within the generic ENTRUST methodology to obtain assurance evidence for particular aspects of self-adaptive systems, they need to be complemented by assurances for other components of a self-adaptive system and integrated in a systematic process as provided by ENTRUST.

Other approaches. We highlight some other related approaches that have been used to provide assurances for self-adaptive systems. Recently, there has been a growing interest in applying control theory to build "correct by construction" controllers [172]. The approach was used to automatically synthesise controllers at runtime, providing control-theoretic guarantees for stability, overshoot, setting time and robustness of system operating under disturbances [75]. Although promising, this research is at an early stage, and its potential to deliver solutions for

Table 4.8: Comparison of ENTRUST to related research on assurances for self-adaptive systems

Approach	Assurance evidence			Assurance argument	Methodology		
	Controller platform	**Controller functions**	**Adaptation decisions**		**Engineering process**	**Tool support**	**Other reusable components**
Generic ENTRUST methodology	Reuse of verified application-independent controller functionality	Verification of controller models to ensure generic controller requirements and some system requirements	Automated synthesis of adaptation assurance evidence during the analysis and planning steps of the MAPE control loop	Development of partial assurance argument at design time, and synthesis of dynamic assurance argument during self-adaptation	Seven-stage process for the systematic engineering of all components of the self-adaptive system, and of an assurance case arguing its suitability for the intended application	Tools specific to each ENTRUST instance	Reusable software artefacts: controller platform, controller model templates; Reusable assurance artefacts: platform assurance evidence, generic controller requirements, assurance argument pattern
Tool-supported ENTRUST instance	Reuse of thoroughly tested virtual machine to directly interpret and run controller models, and of established probabilistic model checking engine	UPPAAL model checking of interacting timed automata models to ensure controller deadlock-freeness, liveness, etc. and functional system requirements	PRISM probabilistic model checking of continually updated stochastic models of the controlled system and the environment to ensure non-functional requirements	Assurance argument synthesised using the industry-adopted Goal Structuring Notation (GSN) standard	Seven-stage process for the systematic engineering of all components of the self-adaptive system, and of an assurance case arguing its suitability for the intended application	UPPAAL used to verify controller models; PRISM used to verify stochastic system and environment models	Reusable controller platform (virtual machine, probabilistic verification engine), timed automata controller model templates; Reusable platform assurance evidence, CTL generic controller require-ments, GSN assurance argument pattern

real-world systems and scenarios has yet to be confirmed. In contrast, ENTRUST relies on proven software engineering techniques for modelling and analysing software systems and assuring their required properties. Runtime verification is a well-studied lightweight verification technique based on extracting information from a running system to detect whether certain properties are violated. For example, sequences of events can be modeled as observation sequences of a Hidden Markov Model allowing to verify the probability that a temporal property is satisfied by a run of a system given a sampled execution trace [179]. Sanity checks are another approach to check the conformance of requirements of adaptive systems. Sanity checks have been used to evaluate the correctness of resource sharing decisions made by a reasoning engine [197]. Approaches such as runtime verification and sanity checks are often supported by established tools. However, these approaches provide only one piece of evidence. Such approaches can also be used by our generic ENTRUST methodology, which supports the integration of assurance evidence from multiple sources in order to continuously generate an assurance case.

Another line of related research (not specifically targetting self-adaptation and thus not included in Table 4.7) is *runtime certification*, proposed in [163] and further developed in [121, 168]. Runtime certification involves the proactive runtime monitoring of the assumptions made in the assurance case, thereby providing early warnings for potential failures. ENTRUST goes beyond the mere monitoring of assumptions, to evolving the arguments and evidence dynamically based on the runtime verification data, particularly for self-adaptive software assurance. ENTRUST also extends existing work on assurance argument patterns [60] by enabling runtime instantiation.

The ENTRUST methodology and the other research summarised in this section also build on results from the areas of configurable software, configuration optimisation, and performance tuning. For instance, symbolic evaluation has been used to understand the behaviour of configurable software systems [161], dedicated support to automatically verify the correctness of dynamic updates of client-server systems has been proposed [102], and specification languages have been devised to help program library developers expose multiple variations of the same API using different algorithms [187]. Research in this area has been applied to realise self-adaptive software systems. For example, it has been used to deal with the problem of configurability of multi-tenant cloud settings by using a game theoretic approach that maximises tenants' preferences satisfaction [82], to find workarounds and add configuration guards to prevent particular failures [181], and to model the variability of cloud systems and identify reconfigurations that meet given criteria using temporal constraints and reconfiguration operations [174]. While these approaches address adaptation of highly configurable systems at runtime, they provide only specific pieces of evidence. Runtime testing as mentioned above (e.g. [78]) is one interesting approach to ensure that such systems continue to execute in a safe and correct manner when adapting to handle changing environmental conditions. Such an approach could be integrated in the generic ENTRUST methodology as part of the analysis phase to provide guarantees about the runtime decision making process of self-adaptation.

Finally, assurance cases and GSN in particular are related to goal modeling. Several approaches exist that provide alternative means to specify goal models for self-adaptive systems; we discuss a representative selection. RELAX offers a textual language that allows requirements to be temporarily relaxed to deal with uncertainty in adaptation [217]. RELAX has been integrated with traditional goal modeling using KAOS [50]. FLAGS provides both crisp goals specified in linear temporal logic and fuzzy goals specified in fuzzy temporal language [13]. Adaptations are triggered by violated goals and the goal model is modified accordingly to maintain a coherent view of the system and enforce adaptation on the running system. Other researchers specify requirements for adaptive systems as two complementary types: awareness requirements and evolution requirements [175]. Awareness requirements indicate the situations that require adaptation and evolution requirements prescribe what to do in these situations. The development of GSN was influenced by the research on goal modeling. Similar to the approaches discussed above, the notation is used to represent and decompose system goals, but in addition to that explicitly incorporate rationale arguments for the decomposition.

The sparsity of Tables 4.6 and 4.7 makes clear that existing approaches are confined to providing correctness evidence for specific aspects of the self-adaptive software. In contrast to existing work on assurances for self-adaptive systems, Table 4.8 shows that ENTRUST offers an end-to-end methodology for the development of trustworthy self-adaptive software systems. Unique to our approach, this includes the development of assurance arguments. The upper part of Table 4.8 shows how the generic ENTRUST methodology covers the whole spectrum of aspects that are required to provide assurances for self-adaptive systems with strict requirements. The lower part of Table 4.8 shows a concrete tool-supported instantiation of ENTRUST and summarises how the various assurances aspects are covered for this instance. Details about the information summarised in the table are provided in Sections 4.4 and 4.5.

4.9 Conclusion

We introduced ENTRUST, the first end-to-end methodology for the engineering of trustworthy self-adaptive software systems and the dynamic generation of their assurance cases. ENTRUST and its tool-supported instance presented in the paper include methods for the development of verifiable controllers for self-adaptive systems, for the generation of design-time and runtime assurance evidence, and for the runtime instantiation of an assurance argument pattern that we devised specifically for these systems.

The future research directions for our project include evaluating the usability of ENTRUST in a controlled experiment, extending the runtime model checking of system requirements to functional requirements, and reducing the runtime overheads by exploiting recent advances in probabilistic model checking at runtime [34, 38, 72, 87, 113]. In addition, we are planning to explore the applicability of ENTRUST to other systems and application domains.

Chapter 5

Model-based Simulation at Runtime for Self-adaptive Systems

In this chapter, we introduce a novel modular approach for decision making in self-adaptive systems that uses runtime simulation to efficiently provide guarantees for the adaptation goals. This chapter is published at Models at Runtime, International Conference on Autonomic Computing, 2016, and copied from [209].

This chapter provides an initial solution to provide guarantees for adaptation goals in an efficient manner. Figure 4.1 shows an overview of the context of the contribution.

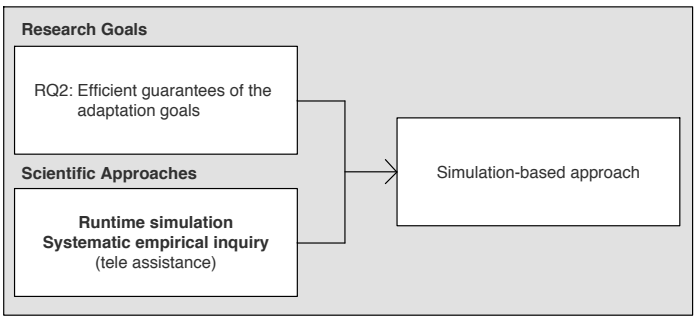


Figure 5.1: Overview of the goals and scientific methods used in this chapter

To address RQ2 (efficiently providing guarantees for the adaptations goals), we propose a novel modular approach that uses simulation and statistical methods to efficiently provide adaptation guarantees for self-adaptive systems. The approach uses distinct quality models for each relevant quality requirement with runtime simulation of the quality models to provide efficient guarantees for the adaptation goals. The approach is evaluated using a simulated tele assistance system.

My contribution to the research presented in this chapter is as follows: I contributed 50% of the conceptualisation, 100% of the technical realisation, and 30% of writing the chapter.

Model-based Simulation at Runtime for Self-adaptive Systems

Abstract

Modern software systems are subject to uncertainties, such as dynamics in the availability of resources or changes of system goals. Self-adaptation enables a system to reason about runtime models to adapt itself and realises its goals under uncertainties. Our focus is on providing guarantees for adaption goals. A prominent approach to provide such guarantees is automated verification of a stochastic model that encodes up-to-date knowledge of the system and relevant qualities. The verification results allow selecting an adaption option that satisfies the goals. There are two issues with this state of the art approach: i) changing goals at runtime (a challenging type of uncertainty) is difficult, and ii) exhaustive verification suffers from the state space explosion problem. In this paper, we propose a novel modular approach for decision making in self-adaptive systems that combines distinct models for each relevant quality with runtime simulation of the models. Distinct models support on the fly changes of goals. Simulation enables efficient decision making to select an adaptation option that satisfies the system goals. The tradeoff is that simulation results can only provide guarantees with a certain level of accuracy. We demonstrate the benefits and tradeoffs of the approach for a service-based telecare system.

5.1 Introduction

Over the past years, various self-adaptation approaches have been proposed to deal with the dynamics and uncertainties of software systems (e.g., the availability of resources or changes of system goals may be difficult to predict at design time). Central to these approaches are feedback loops equipped with models that are updated at runtime, when new knowledge becomes available. The system uses these models to reflect upon itself and achieve its goals by adapting itself in response to changing conditions [48, 133]. With the increasing demand for self-adaptation in applications with critical goals, providing guarantees for the system goals has become an important subject of research [49, 186, 213].

One prominent approach to provide such guarantees is runtime automated verification that allows checking whether certain properties hold for the system model during operation. There is a particular interest in using stochastic models that encode system behaviour and knowledge of relevant qualities. The probabilities of the transitions can be based on value estimates provided by domain experts, but as these values may change over time they need to be updated online [68]. Verification of a stochastic model through exhaustive analysis of the state-transition graph of the system model enables to calculate expected quality properties (e.g., likelihood of failures, expected response times) for different adaptation options, allowing the system to select an option that satisfies the system goals and adapt accordingly [37, 38].

There are two issues with this state of the art approach. Encoding the system behavior and knowledge of different quality properties in a single model lacks flexibility to change goals at runtime, which is an important, but challenging type of uncertainty [133]. Furthermore, exhaustive verification suffers from the state-space explosion problem, which puts constraints on the time and resources required to perform verification, and the size of the models that can be verified. This problem becomes particular relevant for verification at runtime, when time and resources are often constrained. Optimisation techniques have been proposed, for example caching and lookahead [87], but new approaches will be required to provide guarantees for self-adaptation at runtime in an efficient way.

In this paper, we propose a novel modular approach for decision making in self-adaptive systems that is based on distinct models for each relevant quality combined with runtime simulation of the models. Distinct models support on the fly changes of goals. Simulation enables efficient decision making to select an adaptation option that satisfies the system goals. By using statistical techniques, simulation allows to provide results with a required level of accuracy. Simulation is less time and resource consuming compared exhaustive verification approaches. However, the tradeoff is that the guarantees are bounded to a certain level of accuracy.

The remainder of this paper is structured as follows. Section 5.2 discusses selected related work. In Section 5.3, we introduce a telecare system that we use for illustration and evaluation. Section 5.4 introduces the novel modular approach for decision making in self-adaptive systems. In Section 5.5, we evaluate the approach using the telecare system. Section 5.6 wraps up and outlines directions for future work.

5.2 Related Work

We have divided related work in three parts: i) runtime automated verification, ii) simulation in self-adaptive systems, and iii) adaptation goals. We limit the discussion to a selection of representative approaches from the huge body of work that has been developed over the past years.

There is an increasing trend in the use of formal methods at runtime in self-adaptive systems [214]. [68] represents the possible execution flows of a system at runtime with a discrete time Markov chain. The probabilities that represent uncertainties are dynamically updated with a Bayesian estimator. [72] proposes a two-step verification approach: a pre-computation step computes a set of symbolic expressions, which represent satisfaction of the requirements, a verification step then evaluates the formula by replacing the variables with values gathered at runtime. In [37], formally specified requirements are automatically analyzed using runtime model checking techniques to identify and enforce optimal configurations and resource allocations of service systems that are modeled using a Markov model. [91] uses a Markov decision model of the system that enables an interpreter to drive the execution of the system and guarantee the highest utility for a set of quality properties. [74] propose a effective lightweight filtering approach

that learns and continuously updates the transition probabilities of discrete time Markov models of the system. In summary, state of the art proposes to equip the feedback loop with a stochastic model that maintains up-to-date knowledge about the relevant qualities and uncertainties of the system. This model is kept alive and used by automated verification to identify system configurations that comply with the required goals and adapt the system as required.

Simulation allows to explore many different states of the system without being prevented by an infinite (or at least very large) state space. Simulation runs can provide guarantees at different levels of fidelity, based on the level of abstraction of the model used and number of simulation runs applied. Simulation is a commonly used for evaluating novel approaches for self-adaptation [214]. Evaluation can also be used during the engineering process, e.g., to iteratively improve the design of self-adaptive systems, as in [41]. Simulation is rarely used to support decision making at runtime in self-adaptive systems. One example is [173], that presents an approach that automatically builds a dynamic model of a business process to realise service level agreements, while optimizing system resources. The prediction is based on a simulation model whose parameters are tuned at runtime. As recently pointed out [213], simulation offers interesting opportunities to provide guarantees for self-adaptive systems at runtime. However, the approach has not been well studied yet.

A variety of goal models have been proposed to deal with adaptation. We highlight a few representative examples. The RELAX language [216] allows temporal relaxation of requirements to capture uncertainty. FLAGS [13] proposes "crisp goals" specified in linear temporal logic and "fuzzy goals" specified in fuzzy temporal language. [175] distinguishes "awareness requirements" that refer to situations that require adaptation and "evolution requirements" that prescribe what to do in these situations. While several approaches support the operationalisation of adaptation based on goal models, further research is required to support solutions that allow changing adaptation goals on the fly [49, 133].

5.3 Tele Assistance System

The Tele Assistance System (TAS) provides health support to users in their home [37, 204]. Users wear a device that uses third-party remote services from health care, pharmacy, and emergency service providers. Fig. 5.2 shows the TAS workflow that comprises different services. The workflow can be triggered periodically to measure the user's vital parameters and invoke a medical analysis service. Depending upon the analysis result a pharmacy service can be invoked to deliver new medication to the user or change his/her dose of medication, or the alarm service can be invoked, dispatching an ambulance to the user. The user can also invoke the alarm service directly via a panic button.

Multiple service providers provide concrete services for Alarm service, Medical analysis service, and Drug service, abbr. by AS, MAS, and DS respectively. Concrete services have a failure rate F_rate and an invocation $Cost$. Table 5.1 shows the initial values declared by the service providers.

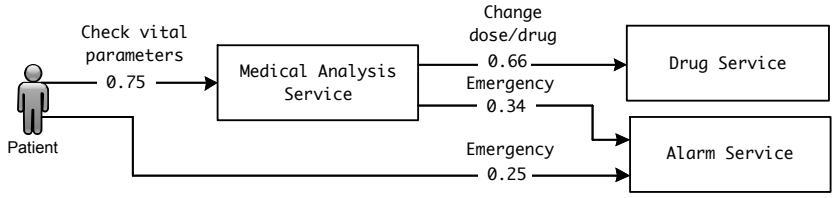


Figure 5.2: TAS workflow

As a default behavior we assume that TAS selects a particular configuration of services, e.g. $\{AS_3,\ MAS_4,\ DS_1\}$. We consider two types of uncertainties in TAS. The first one is related to the actions performed to the system. As shown in Fig. 5.2, we assume that on average 75% of the requests are (automatically triggered) checks of vital parameters and 25% are emergency calls invoked by the user. After checking vital parameters, depending upon the result 66% of the requests invoke the drug service, and 34% of the requests invoke the alarm service. However, these probabilities can change over time. The second uncertainty is related to the concrete services of the system. These uncertainties include the availability of services and quality parameters of running services. Depending upon load on the system, the network and other conditions the initial values of the failure rates and response times of the services are subject to change.

We apply self-adaptation to TAS to deal with uncertainty related to failures, cost, and service time. An offline analysis may find a configuration which supports the set of requirements. But as there are many uncertainties associated with TAS, there is a need for adapting the current configuration at runtime based on the actual values of these uncertainties.

5.4 Modular Decision Making Approach for Self-Adaptation

We introduce the novel modular decision making approach for self-adaptation in two steps. We start with a high level overview of the model for self-adaptation we use in this research. Then we zoom in on change management, the central part of decision making for self-adaptation.

5.4.1 Model for Self-adaptation

In this research, we study architecture-based self-adaptation, where a self-adaptive system consists of a managed system that provides the domain functionality and a managing system that monitors and adapt the managed system [84, 122, 150, 208]. Furthermore, we look at managing systems that are realised with a MAPE-K based feedback loop that is divided in four components: Monitor, Analyze, Plan, and Execute [117, 210], that share common Knowledge (hence, MAPE-K). Knowledge comprises models that provide a causally connected self-representation of

Table 5.1: Third party service profiles for TAS

S.No	AS		MAS		DS	
	F_rate	Cost	F_rate	Cost	F_rate	Cost
1	0.11	4.0	0.12	4.0	0.01	5.0
2	0.04	12.0	0.07	14.0	0.03	3.0
3	0.18	2.0	0.18	2.0	0.05	2.0
4	0.08	3.0	0.10	6.0	0.07	1.0
5	0.14	5.0	0.15	3.0	0.02	4.0

the managed system referring to the structure, behavior, goals, and other relevant aspects of the system [23].

Fig. 5.3 shows the high-level overview of the model for self-adaptation that we use in our research. The model conforms to the three-layer model of Kramer and Magee [122].

The Managed System is the software that is subject of adaptation. At a given time the managed system has a particular configuration that is determined by the arrangement and settings of the running components that make up the Managed System. The set of possible configurations can change over time. We refer to the different choices for adaptation from a given configuration as the *adaptation options*, or alternatively the *possible configurations*. Adapting the managed system means selecting an adaptation option and changing the current configuration accordingly. We assume that the Managed System is equipped with probes and effectors to support monitoring the system and apply adaptations. How these probes and effectors are realised is out of scope of this paper. The Managed System is deployed in an environment that can be the physical world or computing elements that are not under control of the Managed System. The Managed System and the environment may expose stochastic behavior.

The Managing System comprises two sublayers: Change Management and Goal Management. Change Management adapts the Managed System at runtime, using the MAPE components of the feedback loop that interact with the Knowledge Repository. The MAPE components can trigger one another, for example, the Analyser may trigger the Planner once analysis is completed. The Analyser is supported by a Runtime Simulator that can run simulations on the models of the Knowledge Repository during operation. In our current work on modular decision making for self-adaptation, we consider single MAPE loops. Extensions to interacting MAPE loops is subject of our future work. Goal Management enables to adapt Change Management itself. Goal Management offers an interface to the user to change the adaptation logic, for example, to change the models of the knowledge repository, or change the MAPE functions. Changing the adaptation software should be done safely, e.g., in quiescent states [123]. We do not elaborate in these technical aspects as the primary focus of this paper is on Change Management.

Example – TAS is an example of a Managed System and a configuration of this system is an orchestration of a set of concrete services. The adaptation options for TAS are the different combinations of available concrete services. These possible configurations can change over time, e.g., when concrete services are no longer

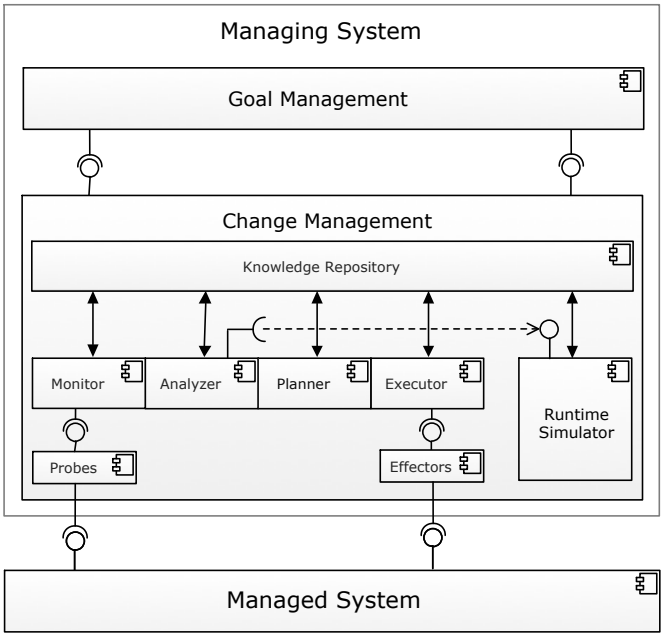


Figure 5.3: High-level model for self-adaptation

available or new services appear. The current configuration can be adapted by replacing one or more concrete services that provide better quality of service. TAS exposes stochastic behavior both with respect to the actions invoked to the system as changes in the quality and other parameters of the concrete services. To deal with the uncertainties, a Managing System is added to TAS that aims to guarantee the system goals regardless of the uncertainties.

We have developed a concrete realisation of the modular approach for decision making in self-adaptive systems that we used for evaluation (see Section 5.5). For additional information, we refer to the project website.[1]

5.4.2 Change Management

Fig. 5.4 shows the key elements of Change Management. We start with explaining the elements of the Knowledge Repository. Then we explain the MAPE components.

5.4.2.1 Knowledge Repository

The Managed System Model and Environment Model capture the essential elements of the managed system and its environment that are needed to make adaptation decisions. The Quality Models capture the characteristics of the different qualities that are subject of adaptation. All the models are parameterised, where

[1] homepage.lnu.se/staff/daweaa/ActivFORMS/Model-based-simulation.htm

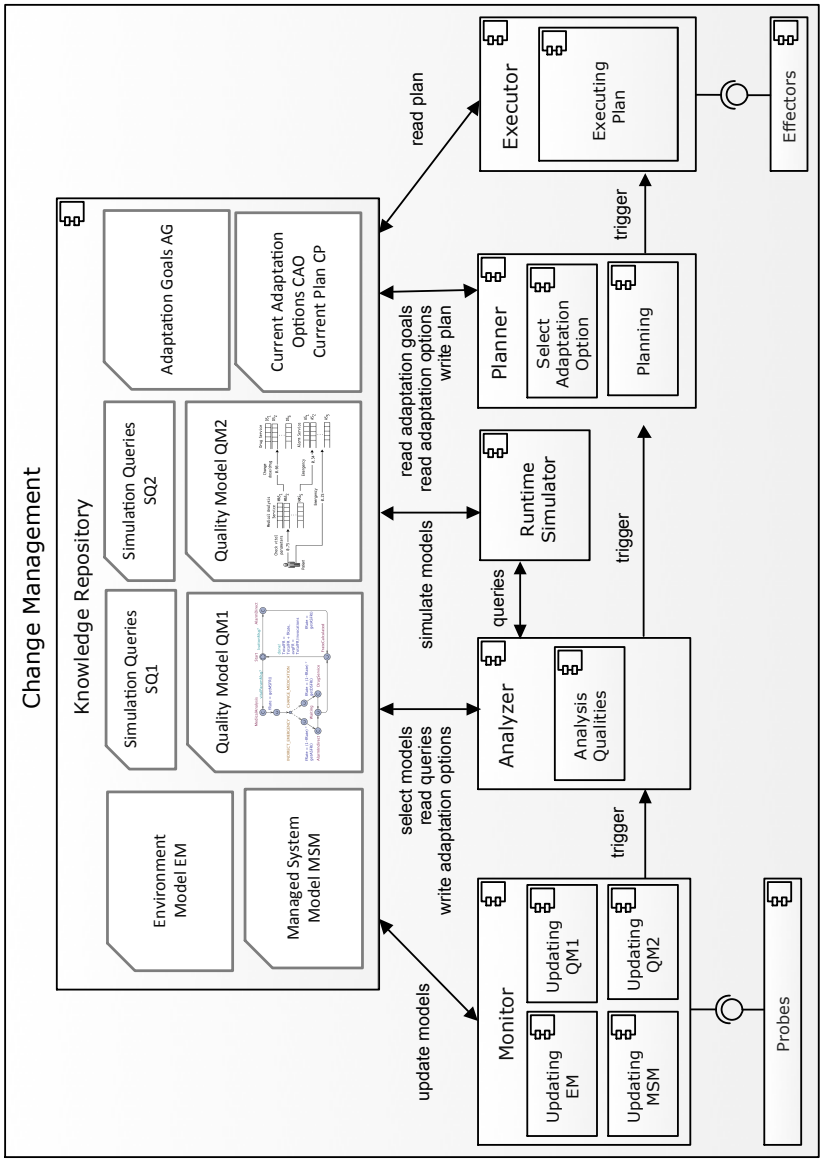


Figure 5.4: Change Management of the modular approach for decision making in self-adaptive systems

the parameters represent variability and/or uncertainty of model elements. A central aspect of the modular approach for decision making in self-adaptive systems is the use of distinct quality models. The approach does not assume any particular types of models; any type of model that supports simulation at runtime (if needed) can be used. In our current realisation, we use stochastic timed automate (STA) as modeling language. STA are a stochastic extension of timed automata [19, 59]. A timed automaton is a finite state machine extended with a set of real-valued clocks. STA allow to represent uncertainties by probabilities associated with transitions in the models. Furthermore, STA models can be parameterized to capture variations or changes in the system or the environment.

For each quality model, a set of Simulation Queries is provided. A simulation query enables determining an estimate for the value of a quality property for a possible configuration by running one or more simulations on the configuration model with the corresponding quality model. A simulation query is formulated as $simulate\ N[<=\ bound]\{E1, .., Ek\}$, where N is the number of simulation runs to be performed, $bound$ is the time bound on the simulation runs, and $E1, ..., Ek$ are state-based expressions that need to be monitored during the simulation. The time is the simulation time, where each tick represents a period of wall clock time. So, a query simulates the system N times over a given period of time to provide insight to the user on the behavior of the system for the expressions $E1, ..., Ek$.

The Adaptation Goals define the objectives that need to be realised by the MAPE components. The modular approach for decision making in self-adaptive systems supports any type of representation of adaptation goals. In our current realisation, we represent adaptation goals as a set of rules defined over the quality properties that are subject of adaptation.

Finally, the Current Adaptation Options list the possible configurations of the managed system that may be ranked based on the adaptation goals. The Current Plan comprises the set of actions that are required to adapt the current configuration to the selected adaptation option.

Example – The Managed System and Environment Model of TAS capture essential aspects of the telecare system and its users. E.g., the environment model represents the behavior of the user, where the preferences for user actions can be expressed as probabilities. The model of the managed system captures the essential elements of the TAS workflow. The concrete services that are used by the system are parameters in the model. In TAS, we use distinct models for failure behavior of services, service times of service invocations, and cost for using TAS. Each quality model is provided with a simulation query that enables determining an estimate for the value of a quality property for a possible configuration. For example, a query to estimate the expected average failure rate of a possible configuration based on 35 simulations, each for a period of 100 time units could be:

$$simulate\ 35[<=\ 100]\{AssistanceService.failureRate\}$$

Examples of adaptation goals in TAS are:

$$R1: averageCost\ <=\ 8\ (\times 10^{-3})$$

$$R2: averageResponseTime\ <=\ 2.5\,s\ (\times 10^{-3})$$

The first rule states that the average cost per invocation should not exceed 8 units per 1000 invocations. The second rule states that the average response time per service invocation should be below 2.5 seconds per 1000 invocations.

The current adaptation options is a list of the possible configurations of the managed system with estimates of the respective qualities. Here is an example entry in this list:

$\{\text{MAS}_1, \text{DS}_4, \text{AS}_5\}$, fRate= 0.12, cost = 8.5, sTime = 17

The current plan in TAS contains a set of required service replacements to transfer the current configuration to the new configuration selected by the decision making mechanism.

5.4.2.2 MAPE Components

The Monitor component tracks the behavior of the managed system and the environment through probes updating the runtime models. The monitor comprises distinct Updating components for each runtime model. The approach does not assume any particular type of updating mechanism. Examples are basic updating mechanisms that update the parameter values of models based on changes in the underlying system or the environment, or more advanced mechanisms such as Bayesian and reinforcement learners.

The Analyzer component analyses the up to date knowledge of the models to determine whether an adaption is required. To that end, the Analyzer uses the Runtime Simulator to estimate the qualities of each possible configuration. Concretely, the Analyzer starts with selecting models (managed system model for a concrete configuration, environment model, and a particular quality model. Then the Analyzer invokes the simulation query for the given quality model. The parameters of the simulation query (N and bound) are configured based on the required accuracy. The simulator uses the selected models to compute an estimate for the quality using the simulation query. This estimate is returned to the Analyzer. In our current research, we use the *standard error of the mean* (SEM) as a measure to determine the accuracy of the simulation queries. The SEM quantifies how precisely a simulation result represents the true mean of the population (and is thus expressed in units of the data). SEM takes into account the value of the standard deviation and the sample size. Concretely, we use the relative SEM (RSEM), which is the SEM divided by the sample mean and expressed as a percentage. For example, a RSEM of 5 % represents an accuracy with a SEM of plus/minus 0.5 for a mean value of 10. Evidently, more accurate results (better estimates) require smaller RSEM values and thus more simulation runs. Currently, we empirically determine the number of simulation runs required for a particular accuracy based on offline experiments. Once the Analyzer has performed an analysis of all the qualities for all the possible configurations, it writes the adaption options to the Knowledge Repository.

The Planner component ranks the adaptation options based on the adaptation goals and creates a plan for the highest ranked option. This plan is then used by the Executor component to adapt the managed system.

Example – The TAS Monitor comprises Updating components for the different runtime models. The preferences of user actions of the environment model are periodically updated based on information directly retrieved from the service providers. The Updating mechanism of the managed system model tracks concrete services that disappear or new concrete services that become available and updates the knowledge accordingly. For the properties of the different quality models (failure rates, response time, queue lengths) we use simple learning algorithms that track the averages of the respective properties over a period of time. The TAS Analyzer uses the Uppaal-SMC engine [59] to perform the simulations of the runtime models. To determine the parameters of the simulation queries we performed a series of offline experiments for different TAS configurations with different qualities and parameter settings. Based on these results, we have set the number of required simulations to 50 for a RSEM of 10% and to 125 for a RSEM of 5%. For details of this experiments, we refer to the project website. The selection of the adaption option for TAS is based on sequentially applying the rules that define the adaptation goals. As an example, for an adaptation scenario that considers failure rates, cost, and service time, first the possible configurations with a failure rate below a required value are selected. From this set the possible configurations with a cost below a certain value are selected. Finally, the configuration with the lowest service time is selected and a plan is generated and executed to adapt the system.

5.5 Evaluation

We now evaluate the modular approach for decision making in self-adaptive systems using a prototype realisation of TAS. We start with presenting the different runtime models that we used in the experiments. Then, we present the results of a first series of experiments in which we consider failure rates and service invocation costs. Next, we extend the first case taking into account the service time of TAS, demonstrating the flexibility of the modular approach. We conclude with experiments that show the scalability of the approach by comparing the adaptation time with an exhaustive approach based on runtime quantitative verification.

5.5.1 Runtime Models

The runtime models of TAS are modeled using stochastic timed automata (STA); an extension of timed automata with stochastic behavior. STA communicate through broadcast signals and shared variables creating networks of STA.

Fig. 5.5 shows the environment model that represents the actions invoked to TAS. We use a scenario where each time tick either a sample of the vital parameters is taken from the user (with a probability value of *p_ANALYSIS*) or the alarm button is pushed by the user (with a probability *p_EMERGENCY*, which is equal to *1 - p_ANALYSIS*). A sample is sent for analysis via the signal *medicalAnalysis!*, while pushing the alarm button triggers an emergency call via the *emergency!* signal. The probabilities are updated at runtime. After invoking an action, TAS processes the request. Once the service completes, the user is notified via the *serviced?* signal.

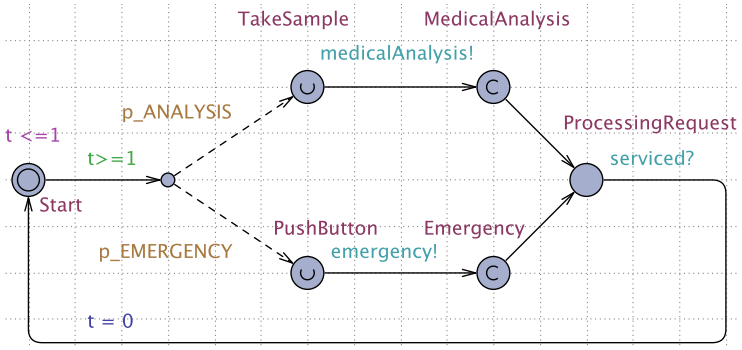


Figure 5.5: Environment model

Fig. 5.6 shows the model of the managed system. The system starts with assigning concrete services to the workflow using the function *assignServices(AD, MAS, DS)* and then waits for incoming requests. The parameters *AD, MAS, DS* can be assigned any concrete instance that is available of the alarm services, medical analysis services, and drug services respectively. Upon receiving a request for medical analysis or an emergency call, respectively the signals *vitalParamMsg!* or the *buttonMsg!* are sent to the workflow model (i.e., a selected quality model as we will discuss below). The managed system model keeps track of the number of invocations, which is required by the quality models to calculate averages. After invoking the workflow the request is processed in the *Processing* state until the *done?* signal is received, which triggers a notification to the Environment model via the *serviced!* signal.

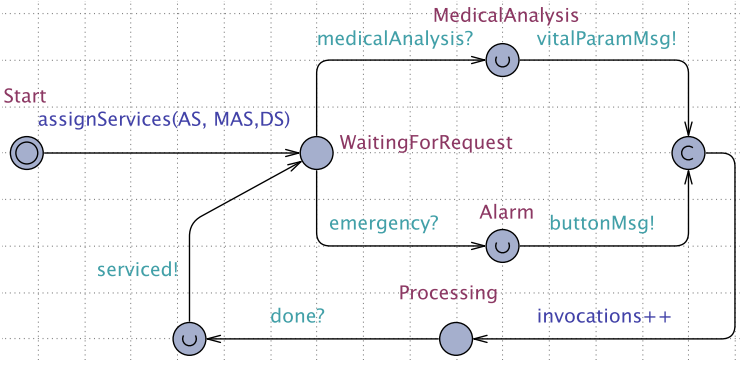


Figure 5.6: Managed System model

Fig. 5.7 shows the quality model of failure behavior of the assistance service of TAS. The model allows estimating failure rates of assistance service invocations. An assistance service invocation fails if any of the services that is needed fails. If the alarm service is directly invoked calculating the failure rate is straightforward and equal to the actual failure rate of the concrete alarm service that is used (*getASFR()*). If the medical analysis service is invoked the failure rate is calcu-

lated by summing the failure rate of the concrete analysis service plus a fraction of the failure rate of the concrete alarm service or drug service, depending on service that is required service (that is, the path that is taken based on the probabilities *p_INDIRECT_EMERGENCY* and *p_CHANGE_MEDICATION*). Finally, the average failure rate is calculated using the total number of invocations. By increasing the number of simulations of the model, the estimated average failure rate will get closer to the real average. The simulation queries to estimate failure rates with an accuracy of RSEM 10% and 5% respectively are:

$simulate1[<= 50]AssistanceService.avgFRate$
$simulate1[<= 125]AssistanceService.avgFRate$

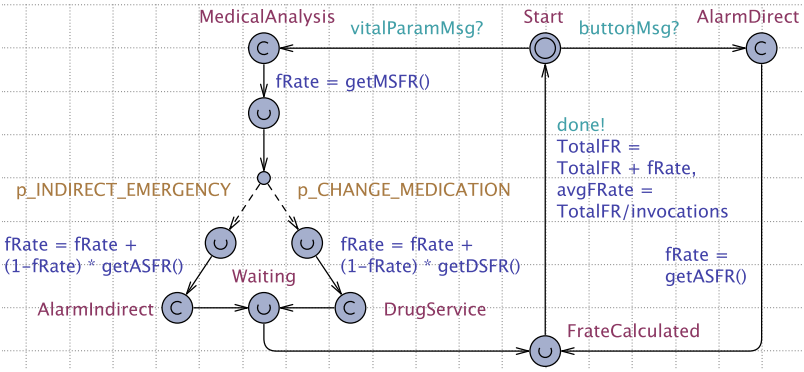


Figure 5.7: Quality model: Failure rate

Fig. 5.8 shows the quality model to calculate estimated costs of using the assistance service. The total cost of an invocation is equal to the sum of the costs of the concrete services used. Similarly to estimating the failure rate, the total cost per invocation depends on the path that is taken in the workflow, based on the probabilities of the actions taken and types of services invoked. The simulation queries to estimate average cost with RSEM 10% and 5% respectively are:

$simulate1[<= 50]AssistanceService.avgCost$
$simulate1[<= 125]AssistanceService.avgCost$

5.5.2 Experiments with Two Qualities

In the first experiment, we focus on two qualities: failure rate and cost. Concretely, adaptation should guarantee the following quality requirements:

R1. $failureRate <= 1.5\ (\times 10^{-3})$

R2. $averageCost <= 8\ (\times 10^{-3})$

R3. *Subject to* $R1$ *and* $R2$ *being satisfied, the* $failureRate$ *should be minimized.*

We use the TAS setting with five concrete instances per service type as described in Section 5.3 and Table 5.1 in particular. We added uncertainty to probabilities of

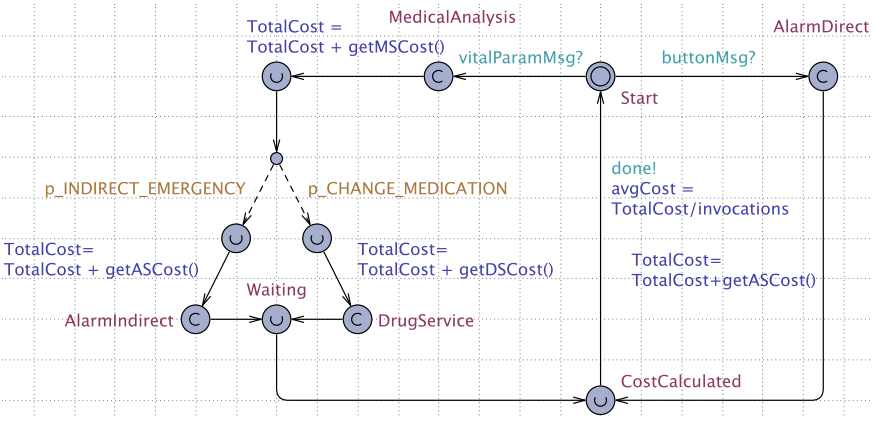


Figure 5.8: Quality model: Cost

the service failure rates and invoked requests based on a normal distribution with a standard deviation of 0.05 and 0.10 respectively. The experiments are performed on a Macbook with 2.5 GHz Core i7 processor, and 16 GB 1600MHz DD3 RAM.

Fig. 5.9 shows the simulation results of all possible configurations of TAS and selected configuration for adaptation at a given point in time. Each configuration is represented by a dot that shows the estimated values for failure rate and average cost of that configuration. The rectangle area demarcated by the dotted lines contains all the possible configurations that comply with requirements $R1$ and $R2$. Based on requirement $R3$ the configuration with the lowest failure rate is selected for adaption. Note that the dot for each configuration is an estimate with an accuracy that is based on the simulation query used, in this particular case a query with accuracy of SEM 5%.

Fig. 5.10 shows how the change of estimated quality properties over time due to uncertainties, incl. changes in the user behavior and the quality properties. We can see that the cost requirement ($R2$) of the initially selected configuration in Fig. 5.9 (Config 1 in Fig. 5.10) is violated. Hence, another configuration is selected now (Config 2). This figure underpins the importance of adaptation at runtime.

We now discuss the results of adaptation. Fig. 5.11 shows the result of a series of 10000 invocations of the assistance service with simulation queries of RSEM 5% and 10%. The managed system checks for adaptation every 500 invocations. Aligned with requirements $R1$ and $R2$, we have calculated the failure rates and average costs over a moving window of 1000 invocations. The boxplots for failure rate show that the required value of 0.15 is satified. The boxplots for the average cost show that the cost remains below the required 8 units at all times with some exceptions for RSEM of 10%. The boxplots for adaptation times show the tradeoff of getting stronger guarantees based on different accuracy levels, i.e., RSEM of 5% takes almost double the adaptation time as RSEM of 10%; the major part of this time is used for runtime simulation to estimate the quality properties of the adaptation options.

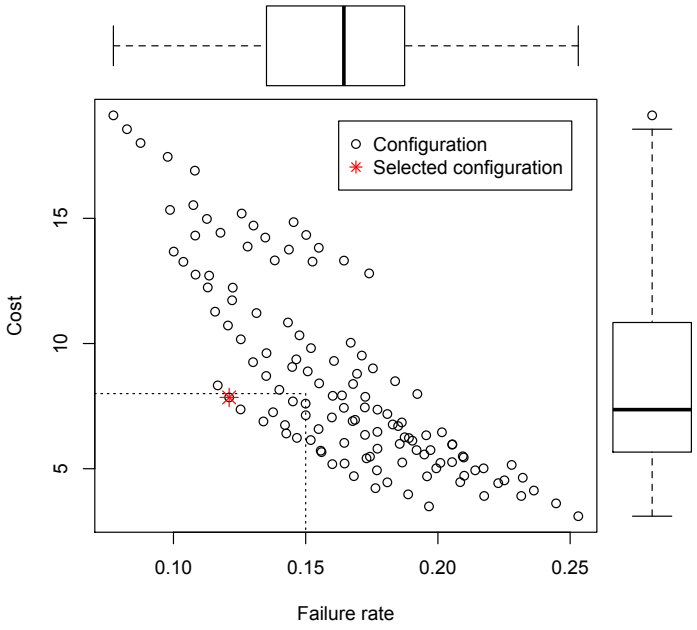


Figure 5.9: Adaptation options with the selected configuration

5.5.3 Experiments with Three Qualities

We now demonstrate the flexibility of our approach by adding additional goals and rules. Concretely, we add a new quality property that is subject of adaptation: service time. Service time comprises two components: the response time of invocations of concrete services and the waiting time due to queues with pending invocations. Service time is an important concern in TAS as users may need to get treatments quickly.

Table 5.2 shows initial estimated response times (in sec) and queue lengths (pending invocations) for the concrete services.

Table 5.2: Average queue lengths and response times

S.No	AS		MAS		DS	
	Rtime	Qlen.	Rtime	Qlen.	Rtime	Qlen.
1	5.7	3	11.0	1	8.0	1
2	7.3	2	9.4	4	7.7	3
3	3.8	5	20.0	2	11.0	5
4	9.5	1	8.0	6	10.0	2
5	18.6	4	9.0	3	15.0	4

For the adaptation goals, we replace $R3$ as follows:

R3'. *Subject to R1 and R2 being satisfied, the serviceTime should be minimized.*

To realise this new requirement, we need to add a new quality model to the knowledge repository, update the adaptation goals, add an Updating component for

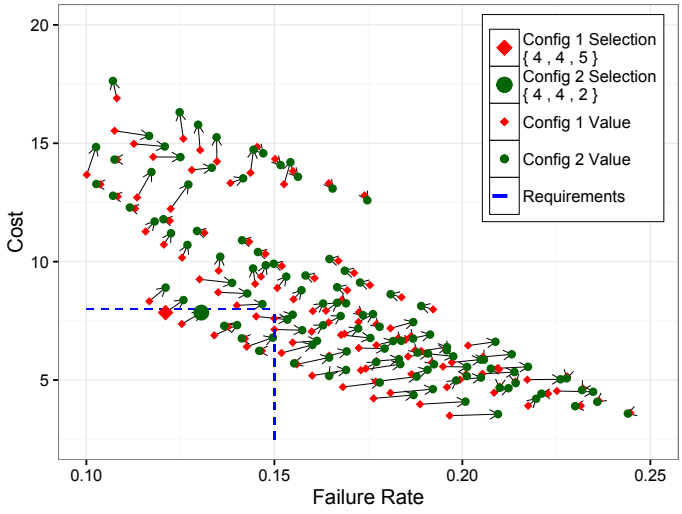


Figure 5.10: Differences between two configurations

the new quality model in the Monitor component and extend the decision making logic to handle the new quality. Our current realisation supports such updates on the fly, based on runtime executable formal models [79, 109] (see also the project website).

Fig. 5.12 shows the quality model to estimate service times. The service time per invocation is accumulated by the time the request has to wait in the queues plus the actual execution time, depending on the path that is taken.

For simulation, we used the following query:

$$simulate1[<= 50]AssistanceService.avgSTime$$

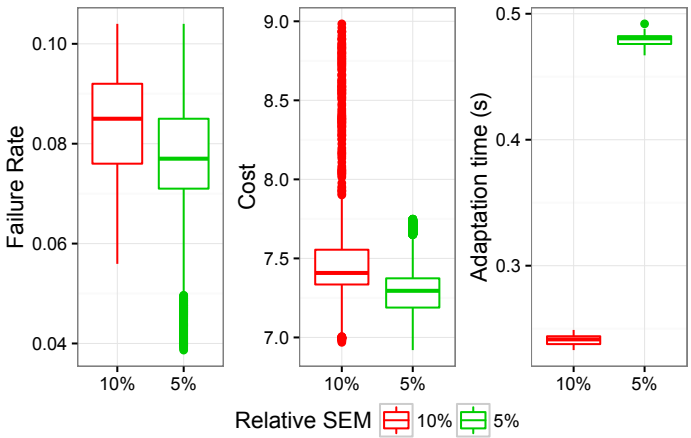


Figure 5.11: Results over 10000 runs for the first experiment

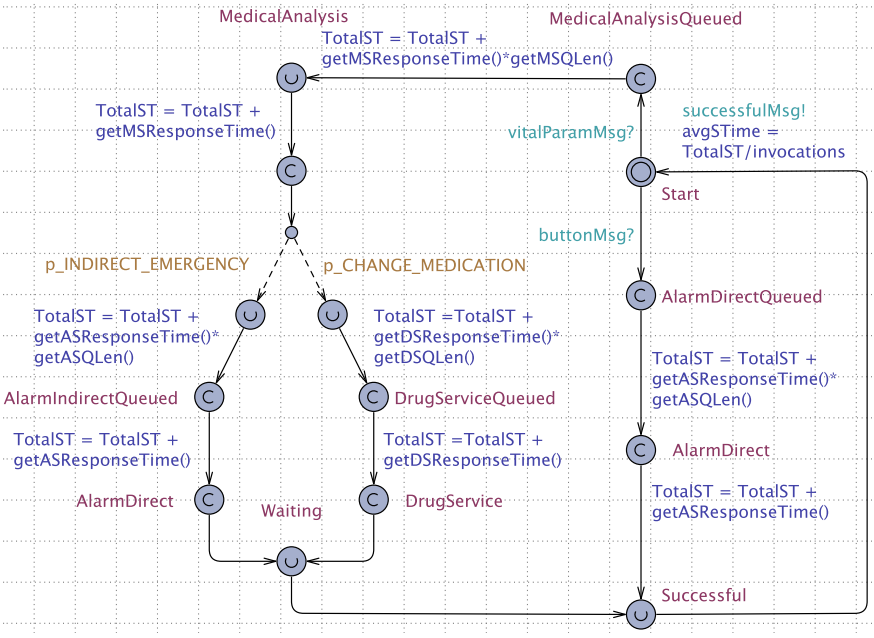


Figure 5.12: Quality model: Service time

$$simulate1[<= 125]AssistanceService.avgSTime$$

Fig. 5.13 shows the simulation results of all possible configurations at a given point in time and the configuration that is selected for adaptation. Among all the configurations that comply to $R1$ and $R2$ (valid configurations), the one with the lowest service time is selected for adaptation.

Fig. 5.14 shows the result of 10000 invocations of the assistance service for RSEM of 10% and 5%. As for the first experiment, we show the quality properties for a sliding window of 1000 invocations. The boxplots for failure rate show similar results for RSEM 5% and 10%; both realise $R1$. The boxplots for cost show that RSEM of 5% gives slightly better results, but RSEM of 10% violates $R2$ some times. The boxplots for service times are similar. Similar to the first experiment, the boxplots for adaptation times show that RSEM of 5% takes almost double the time as RSEM of 10%.

5.5.4 Scalability

To conclude, we compare the scalability of our approach with Runtime Quantitative Verification (RQV), an exhaustively verification technique. For RQV, we used a minimal discrete time Markov model of TAS and used PRISM for verification of quality properties. We used the same setup as in first experiment and systematically increased the number of concrete services per service type. The probabilities of actions and quality properties are assigned randomly. Fig. 5.15 compares the time required for adaptation. The graph shows that the modular approach is signif-

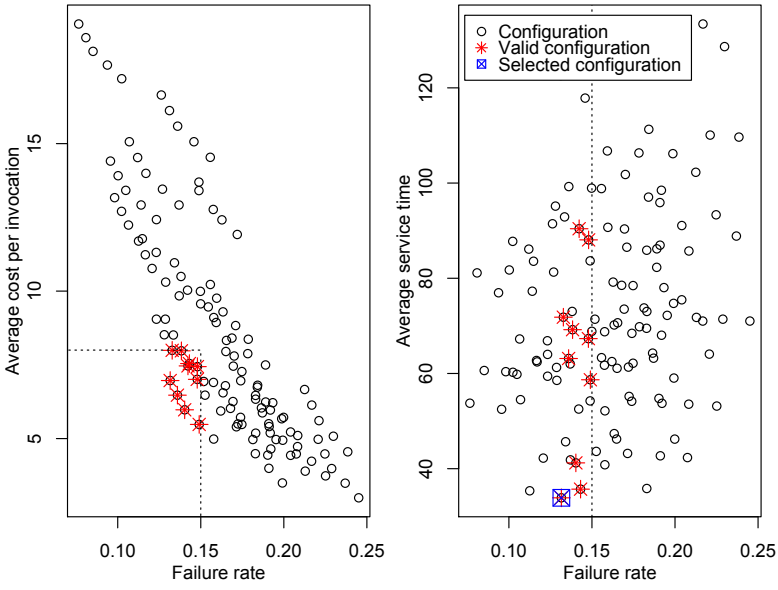


Figure 5.13: Adaptation options with selected configuration

icantly faster than RQV. On the other hand, RQV guarantees the required qualities, while the accuracy of simulation is bound to the selected RSEM. Additional test are required to further compare the tradeoffs of both approaches.

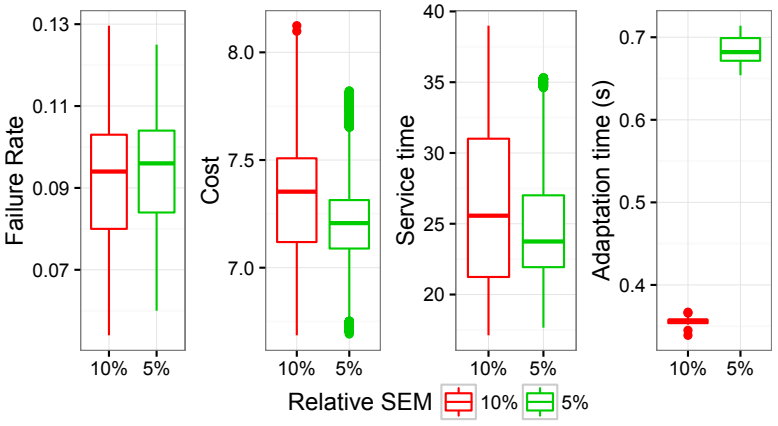


Figure 5.14: Results over 10000 runs for the first experiment

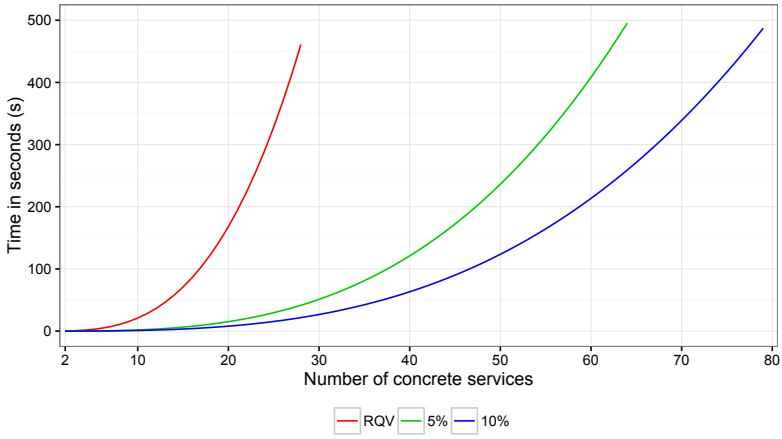


Figure 5.15: Comparison of scalability with RQV

5.6 Conclusion and future work

This research contributes a novel modular approach for decision making in self-adaptive systems. The approach is based on distinct runtime models for different qualities supporting on the fly changes of quality models and adaptation goals. As the approach uses simulation to estimate required quality properties, it is inherently more efficient compared to exhaustive approaches. This is confirmed by initial test results. However, the consequence of using simulation is a reduction of accuracy, which may lead to temporal violations of requirements. On the other hand, the approach allows to tradeoff the accuracy provided by the time that is required to adapt. In the future, we are planning an depth comparison between the proposed approach and exhaustive approaches. We also plan to extend the type of queries by studying how we can use statistical model checking at runtime to support efficient decision making in self-adaptive systems.

Chapter 6

ActivFORMS: An Efficient Approach to Engineer Self-Adaptive Systems with Guarantees

In this chapter, we present the integrated ActivFORMS approach that provides guarantees for the functional correctness of the feedback loop, efficient guarantees about the adaptation goals, and support for changing adaptation goals on the fly. This chapter is currently under review and copied from [205].

In this chapter, we take a holistic approach to engineer self-adaptive systems with ActivFORMS. Figure 6.1 provides an overview of the research goals and scientific approaches used in this chapter.

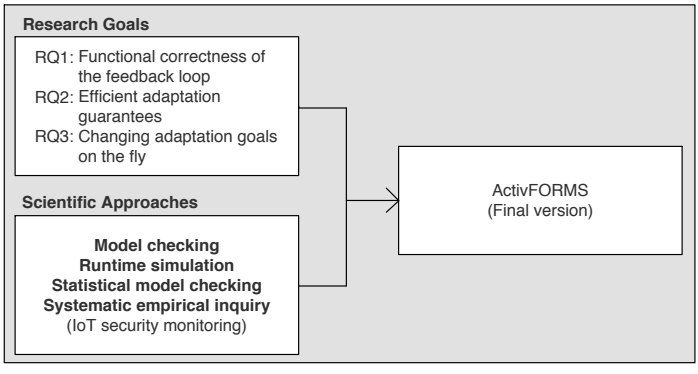


Figure 6.1: Overview of the addressed research goals and used scientific approaches in this chapter

ActivFORMS offers a set of templates to model and verify feedback loop models at design time. The verified models of the feedback loop are directly executed to ensure correctness of the feedback loop (RQ1). Efficient guarantees for the adaptation goals are provided using runtime simulation and statistical model checking (RQ2). The ActivFORMS runtime supports on the fly changing of adaptation goals and the feedback loop model (RQ3).

My contribution to the research presented in this chapter is as follows: I contributed 60% of the conceptualisation, 100% of the technical realisation, 100% of the evaluation, and 20% of writing the chapter.

ActivFORMS: An Efficient Approach to Engineer Self-Adaptive Systems with Guarantees

Abstract

Self-adaptation provides a principled way to deal with uncertainties at runtime. A self-adaptive system employs a feedback loop to continuously monitor and adapt itself to achieve particular quality goals (i.e., adaptation goals) regardless of uncertainties such as changes in operating conditions or user requirements. Guaranteeing compliance of the adaptation goals is challenging since the uncertainties can appear at any time. Recent research suggests the use of formal techniques at runtime to provide guarantees for adaptation goals. For example, a self-adaptive Internet of Things (IoT) application tracks network interference to update a parameterised Markov model of the system and verifies the updated model to identify and apply a configuration with minimal packet loss (i.e., the adaptation goal). Such approaches primarily focus on ensuring the required quality of service, typically using exhaustive verification. In this paper, we take a broader perspective and look at guarantees for the functional correctness of the feedback loop, the efficiency of providing guarantees for the adaptation goals, and support for dealing with changing adaptation goals at runtime. To that end, we present ActivFORMS (Active FORmal Models for Self-adaptation), a formally founded model-driven approach for engineering self-adaptive systems. ActivFORMS provides: 1) functional correctness of the feedback loop by direct execution of formally verified models of the feedback loop using a reusable virtual machine, 2) efficient guarantees for the adaption goals with a required level of confidence using statistical model checking techniques at runtime, and 3) support for changing adaptation goals on the fly and updating of verified models of the feedback loop that meet the new goals. We demonstrate our approach for a real world IoT building security monitoring application deployed at KU Leuven.

6.1 Introduction

The increasing pervasiveness, connectivity, and mobility of computing devices introduce uncertainties that pose severe challenges to software engineers. Providing the required levels of quality, such as performance, reliability, and efficiency becomes particularly challenging for systems that must operate with dynamic available resources, goals that change, etc. These dynamics are often difficult to predict at design time, hence, guaranteeing the required qualities can only be achieved at runtime when the knowledge becomes available that is required to resolve the uncertainties. Self-adaptation is widely considered as an effective approach to cope with this complexity [48, 65, 133, 203]. Self-adaptation equips a system with a feedback loop that tracks changes in the system and the environment, reasons about the changes, and adapts the system to maintain the system goals or degrade gracefully when necessary.

In this research, we focus on architecture-based adaptation, which provides a suitable level of abstraction and generality to handle system dynamics that involve adaptation of components and their relations [84, 122, 150]. Central in architecture-based adaptation is the separation between the domain concerns, which are dealt with by the managed system (that is subject to adaptation), and the adaptation concerns, which are dealt with by the managing system (by adapting the managed system). A well-known approach to structure the managing system is by means of a feedback loop divided into four components: Monitor, Analyze, Plan, and Execute [117]. These components share common Knowledge (hence, MAPE-K) that may contain data about the managed system, the environment, the adaptation goals, and other working data that is shared among the MAPE components [211].

One of the key challenges in engineering self-adaptive systems is providing guarantees that the adaptation goals are being achieved. This is especially important for systems with strict goals. The basic underlying problem is that due to uncertainties, these guarantees need to be delivered during the system's entire lifetime, from inception to and throughout operation. Over the past decade, a variety of approaches have been proposed to provide such guarantees, ranging from formal proof before deployment to testing at runtime [49, 185, 213, 214]. Our focus here is on approaches that apply formal modeling and verification to realise self-adaptation, which is the most popular approach studied so far [48, 133]. A typical approach is described in [112], where the authors create formal models for adaptive systems, verify the models and automatically translate the models into executable programs. The approach guarantees conformance between the models and programs using model-based testing. Another typical example is presented in [37], where quality goals of service-based systems are expressed as probabilistic temporal logic formulae, which are then automatically analysed at runtime using model checking techniques to identify and enforce optimal system configurations.

These representative examples show that the state of the art typically focusses on ensuring the required quality of service. Guarantees for functional correctness of the behaviour of the feedback loop components (i.e., the MAPE elements) and their interactions is often ignored. Consequently, important properties are not evaluated, such as: does the monitor component update the knowledge correctly, does the analysis component correctly identify deviations of the adaptation goals based on the monitored data, or does the execute component perform the adaptation actions of the plan in the correct order? Guaranteeing such properties is important to assure proper adaptation capabilities. Furthermore, existing approaches based on the application of formal techniques at runtime typically rely on exhaustive verification, which is known to be computationally very demanding. Such techniques are problematic in resource-constrained settings, such as for example the IoT. Finally, although runtime changes of goals is considered a very important type of uncertainty [122, 165, 175], so far research on engineering self-adaptive systems has not given sufficient attention to changing adaptation goals on the fly.

In this paper, we present ActivFORMS (Active FORmal Models for Self-adaptation), an integrated formally founded and model-driven approach for engineering self-adaptive systems. ActivFORMS tackles three limitations of current

self-adaptation approaches: functional correctness of the feedback loop, efficient decision making at runtime with guarantees, and support for on the fly changes of adaptation goals. The contributions of ActivFORMS to the state of the art are:

1. ActivFORMS guarantees functional correctness of the feedback loop by direct execution of formally verified models of the feedback loop using a reusable virtual machine;
2. ActivFORMS provides efficient guarantees for the adaption goals with a required level of confidence using statistical model checking techniques at runtime; and
3. ActivFORMS supports changing adaptation goals on the fly and updating the verified models of the feedback loop that meet the new goals.

We evaluate ActivFORMS and compare it with a state of the art approach for a real world IoT building security monitoring application deployed at KU Leuven.

The research presented in this paper is based on a number of assumptions. In line with related research, see e.g. [37, 73, 164], we assume that the managed system already exists and focus on its enhancement with self-adaptation capabilities through the addition of a MAPE-K feedback loop. We assume that the managed system has a limited, but potentially high number of possible configurations (adaptation options) that can dynamically change over time. This implies that system parameters with a continuous domain that determine configurations need to be discretised. We also assume that the managed system is equipped with basic infrastructure for consistent adaptation (changing components, reconfigurations, etc.), for which we can rely on existing solutions. Furthermore, we consider cooperative systems with shared goals. Finally, we target systems for which dynamics in the environment are significantly slower than execution of adaptations and communication. These assumptions hold true for a large class of systems in which software is coordinating or controlling entities that have an explicit location, including systems with mobile nodes. Out of scope are real-time and competitive systems (entities that pursue their own goals). These systems require dedicated solutions (e.g., real-time operating systems) or pose specific security challenges (e.g., establishing trust among elements).

The research presented in this paper leverages on initial work on ActivFORMS. [109] focussed on functional correctness of feedback loops, while the work presented in this paper provides guarantees both for functional and quality properties of self-adaptive systems. [209] explored the possibility to provide guarantees for quality goals using simulation at runtime. In [40], an earlier version of ActivFORMS was used to provide guarantees for the functional correctness of MAPE-based feedback loop models; guarantees for quality goals in that work was based on runtime quantitative verification. In this paper, we provide guarantees for quality goals by applying statistical model checking at runtime. Furthermore, this paper contributes a set of templates to design and verify MAPE-K feedback loops that substantially extend and refine earlier versions [79]. Last but not least, in this paper we provide full support for life updates of adaptation goals and MAPE-K models.

Note also that our work on FORMS [211] is very different from ActivFORMS (despite the similarity of the names). FORMS defines a reference model for self-adaptive systems specified in the Z language, comprising three perspectives on self-adaptation: reflection, MAPE, and distribution. The only correspondence between FORMS and ActivFORMS is that ActivFORMS respects the separation of concerns defined in the reflection perspective of FORMS and applies the basic structure defined by its MAPE perspective. However, different from the reference model defined by FORMS, ActivFORMS offers a concrete approach for engineering self-adaptive systems with guarantees for adaptation goals regardless of uncertainties the system is subject to.

This paper is structured as follows. In Section 6.2 we provide background on timed automata and statistical model checking. Section 6.3 introduces a self-adaptive IoT system that we use to illustrate and evaluate our research. In Section 6.4, we present ActivFORMS that comprises four stages: model and verify feedback loop, deploy feedback loop model with virtual machine, runtime verification of adaptation goals and decision making, and evolution of adaptation goals and feedback loop model. In Section 6.5, we evaluate ActivFORMS using the system introduced in Section 6.3. Section 6.6 discusses related work on formal approaches to self-adaptation and positions ActivFORMS in this landscape. Finally, we draw conclusions and outline directions for future work in Section 6.7.

6.2 Preliminaries

In this section, we provide background information on timed automata and statistical model checking and introduce basic terminology and concepts used in the rest of the paper.

6.2.1 Timed Automata

A timed automaton (or behaviour) [4] is a finite state machine extended with a set of real-valued clocks that progress synchronously. Ordinary variables can be read, manipulated, and written as usual. Automata can be connected forming networks of timed automata. The state of the system is then defined by the state of all automata, the clock values, and the values of the ordinary variables. Only one state per automaton, called *control* or *active* state (or current location), is active at a time. Automata can synchronise through channels, which can be binary or broadcast channels. For a binary channel, a sender $x!$ can synchronize with a receiver $x?$ through a signal x. The sender will be blocked if there is no receiver. A broadcast channel sends a signal to all the receivers; if there is no receiver, the sender will continue. The edge of the automaton can be annotated with: a *guard*, expressing a condition on the values of clocks and variables that must be satisfied for the edge to be taken (e.g. $y < 5$); a *synchronization* action (e.g. *x!*) which, when the edge is taken, forces a synchronization with other components on a complementary action (e.g. *x?*); and an *update* defining actions to be taken when a transition is made (e.g. a function $reset()$ resets clock y to 0). The absence of a guard is interpreted as the condition $true$.

Uppaal [16] offers a model checking suite that supports modeling of behaviors and verification of properties. Behavior specifications can be complemented with expressions specified in a C-like language to define data structures (*struct* concept) and functions. Goals can be expressed in timed computation tree logic (TCTL). TCTL expressions describe state and path formulae that can be verified, such as reachability (a system should/can/cannot/... reach particular states), liveness (something eventually will hold), etc. Uppaal defines two types of transitions between states: *action transition* and *delayed transition*. Action transitions can be further divided into *synchronization transition* and *internal transition*. In a synchronization transition automata synchronize via a channel as explained above. In an internal transition, an automata moves from its current state (say *S*) to a next state (say *T*) via an edge (e) when the conditions hold to make the transition, e.g. the guard on the edge is satisfied, the invariant of *T* holds, etc. In a delayed transition only the clocks tick and no actual state transition is made (e.g. *S* remains active in the controller while $y <$ *MAX_TIME*). Further progress in time might lead to an invariant violation ($y \geq$ *MAX_TIME*) triggering a transition ($S \rightarrow T$). Finally, to enable modelling of atomicity of transition sequences (i.e. multiple transitions with no time delay) states may be marked as *committed*. Committed states (marked with a C in the location) are taken without time delay. Urgent states (marked with a U) are similar, but have a lower priority as committed states.

To support stochastic behaviours, a stochastic interpretation of the timed automata has been proposed [59]. In short, the stochastic interpretation replaces the non-deterministic choices between multiple enabled transitions by probabilistic choices based on the probability distributions. Uppaal supports modelling and verification of networks of stochastic timed automata.

6.2.2 Statistical Model Checking

Statistical model checking (SMC) has been proposed as an efficient alternative to traditional model checking that exhaustive travers all the states of the system [129]. The central idea of SMC is to check the probability $p \in [0, 1]$ that a model M satisfies a property φ, i.e., to check $P_M(\varphi) \geq p$ by performing a series of simulations. SMC applies statistical techniques on the simulation results to decide whether the system satisfies the property with some degree of confidence. To verify a quality property it has to be formulated as a verification query.

Uppaal-SMC [59] is a tool for statistical model checking that supports different types of verification queries; here we focus on probability estimation and simulation. Probability estimation computes an estimation of probability p for an expression φ with an approximation interval $[p-\epsilon, p+\epsilon]$ and confidence $[1-\alpha]$ in a given time $bound$. A probability estimation query is formulated as $p = Pr[bound](\varphi)$. Simulation performs N simulations of the system model in a time $bound$ to provide insight in the values of expected system behaviors. A simulation query is formulated as $simulate\ N[\leq\ bound]\{E1, ..., Ek\}$, where N is the number of simulations to be performed and $E1, ..., Ek$ are the (state-based) expressions that need to be monitored during the simulation. A benefit of SMC is that ϵ, α, N, and $bound$ are parameters that allow designers to tradeoff the accuracy of the results with the resources and the time required for verification. More accurate results

require more resources and verification time, and vice versa. However, in contrast to exhaustive approaches (such as runtime quantitative verification [37]), a simulation-based approach does not provide 100% guarantees, but an estimation that is bound to a confidence interval [51, 59, 219].

In this research, we do not consider rare events for which specific techniques such as importance sampling and importance splitting can be applied to statistical model checking [131].

6.3 Self-Adaptive Internet of Things Application

This section introduces a self-adaptive system, called DeltaIoT, that we use to illustrate ActivFORMS in Section 6.4 and to evaluate the approach in Section 6.5. DeltaIoT system is a reference Internet-of-Things (IoT) application that enables evaluating a new self-adaptation approach and comparing its effectiveness with other solutions [111]. DeltaIoT comprises a simulator for offline experimentation and a physical setup that can be accessed remotely for experimentation in the field. DeltaIoT is part of the smart campus initiative by imec-DistriNet, KU Leuven.[1]

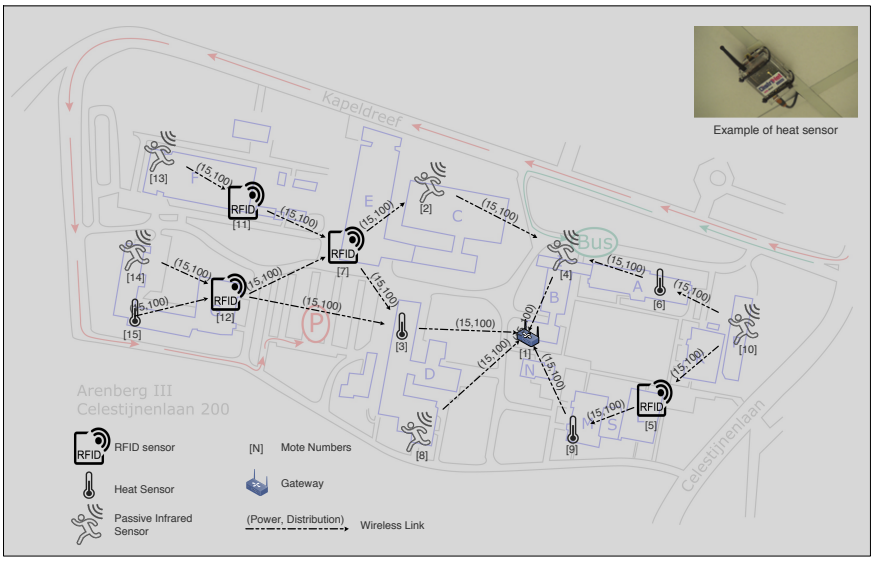


Figure 6.2: DeltaIoT system with network topology

DeltaIoT consists of a collection of 15 LoRa-based IoT motes[2] deployed in various buildings at the KU Leuven campus, see Fig. 6.2. In each building, motes are strategically placed to provide access control to labs (via RFID sensor), to monitor the occupancy status (via Passive infrared sensor) and to sense the temperature (via heat sensor, an example is show top right of Fig. 6.2). The sensor data from all the motes are relayed to the IoT gateway, which is deployed at a central monitoring

[1] https://people.cs.kuleuven.be/ danny.weyns/software/DeltaIoT/
[2] https://www.lora-alliance.org/What-Is-LoRa/Technology

facility. Campus security personnel monitor the status of various buildings and labs from the monitoring facility and take appropriate action whenever unusual behaviour is detected in the buildings.

DeltaIoT uses wireless multi-hop communication. As shown in Fig. 6.2, each IoT mote in the network relays its sensor data to the gateway. However, some of the IoT motes, which are farther away from the gateway, have to relay their sensor data via intermediate IoT motes. DeltaIoT uses time synchronised communication [66]. The communication in the network is organised in cycles, each cycle comprising a fixed number of communication slots. Each slot defines a sender mote and a receiver mote that can communicate with one another. The communication slots are fairly divided among the motes. For example, the system can be configured with a cycle time of 536 second (9.5 minutes) with each cycle comprising 268 slots, each of 2 seconds. For each link, 40 slots are allocated for communication between the motes.

Each mote is equipped with three queues: *buffer* collects the messages produced by the mote, *receive-queue* collects the messages from the mote's children, and *send-queue* queues the messages to be sent to the parent(s) during the next cycle. The size of the *send-queue* is equal to the number of slots that are allocated to the mote for communication. Before communicating, the messages of the *buffer* are first moved to the *send-queue*. The remaining space is then filled with messages from the *receive-queue*. Messages that arrive when the *receive-queue* is full are lost (i.e., queue loss).

IoT applications are expected to last a long period of time on a single battery (e.g. 10 years), while offering reliable communication with minimal latency. To guarantee these quality properties, the motes of the network should be optimally configured. Two key factors that determine the critical quality properties are the transmission power of the motes and the selection of the parents to send messages towards the gateway (more specially, the distribution of the messages sent via the links to the respective parents). Guaranteeing the required quality properties is complex as the system is subject to various types of uncertainties. Here, we consider two primary types of uncertainty:

1. *Network interference and noise:* Due to external factors such as weather conditions and the presence of other WiFi signals in the neighbourhood the quality of the communication between motes may be affected, which in turn may lead to packet loss.
2. *Fluctuating load of messages:* The traffic load of messages produced by the motes may fluctuate in ways that are difficult to predict (e.g., messages produced by a passive infrared sensor are based on the detection of motion of humans).

Experiments in DeltaIoT run default for a period of 24 hours real time. Some motes generate a constant number of messages over time, e.g. temperature sensors, while others generate messages based on context, e.g. the presence of people. In simulation mode, we use profiles for the uncertainties. The data of these profiles was collected from field observations for a period of one week. The graph on the

left hand side of Fig. 6.3 shows the fluctuations of the traffic load generated for one of the motes in Fig. 6.2; the graph on the right hand side shows changes of the signal to noise ratio of the communication path between two motes. Signal to Noise Ratio (SNR) represents the ratio between the level of desired signal and the level of the undesired signal, i.e. noise, which comes from the environment in which the IoT system operates. The higher the interference, the lower the SNR, resulting in higher packet loss. As an alternative for specific profiles, the traffic load for motes can also be determined probabilistically (e.g. $p = 0.5$ generates a traffic load with a probability 0.50), and similarly for the SNR, a value can be picked stochastically in a range $\pm 5.0dB$.

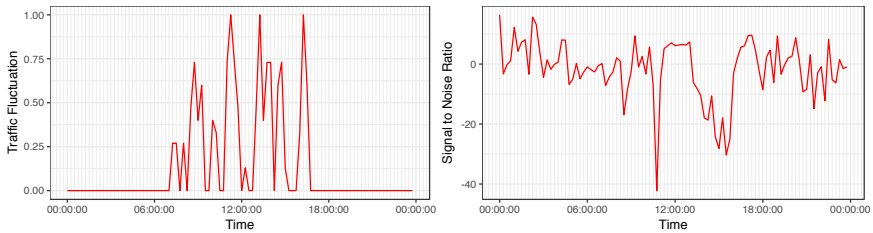


Figure 6.3: Profiles of uncertainties for one of the motes in Figure 6.2.

The quality requirements for DeltaIoT that become adaptation goals for self-adaptation are:

$R1$: The average packet loss[3] should not exceed 10%.

$R2$: The energy consumption should be minimized.

In addition, the following adaptation goal should be added during execution:

$R3$: The average latency of messages should be less than 5% of the cycle time.

DeltaIoT offers a client that comprises a Java package with Probe and Effector classes. Listing 6.1 lists the methods of the probe that can be used to monitor the IoT network and the effector to adapt the mote settings (both in the physical network and the simulator).

Listing 6.1: DeltaIoT probe and effector methods.

```
ArrayList<Mote> getAllMotes();
ArrayList<QoS> getNetworkQoS(Period);
void setMoteSettings(MoteID, List<LinkSetting>);
void resetDefaultConfiguration();
```

$getAllMotes$ returns an array with a representation of each mote of the network for a cycle, including the traffic generated by a mote, the energy consumed, the settings of the transmission power that a mote used to communicate with each of its parent, the spreading factor used for each link,[4] the SNR for each link, and the

[3] The average is defined over a period of 24 hours.

[4] Spreading Factor is defined as the number of chirps used per symbol, where the chirp rate is equal to the bandwidth [8]. A higher spreading factor results in longer range but at the cost of more energy consumption.

distribution factor per link being the percentage of the messages sent by a source mote over the link to each of its parents.[5] $getNetworkQoS$ returns statistical data about the quality of service (QoS) of the overall network for a given period. Currently this method returns data about packet loss, energy consumption, and latency of the network.

$setMoteSettings$ can be used to set the parameters for the parent links of a mote with a given ID. A $LinkSetting$ contains the source and destination node of the link, the transmission power to be used to communicate via the link, and the distribution factor for the link. Finally, $resetDefaultConfiguration$ resets the network settings to predefined values. This method can be used to bring the system to a well-known state, e.g. as failsafe state.

6.4 ActivFORMS Approach

ActivFORMS offers an integrated approach to realise self-adaptation with guarantees. The approach is based on monitor-analyse-plan-execute (MAPE) feedback loops, which are the most common type of feedback loop used to engineer self-adaptive software systems, see e.g. [37, 65, 117, 210]. Other types of adaptation, e.g. based on principles from control theory (for a recent survey see [172]) are not supported by ActivFORMS. The foundational principles of ActivFORMS are:

1. Model-driven: models are the central artifacts to achieve the adaptation goals from design to operation, using model-based analysis, formal verification, and direct model execution.
2. Continuous guarantees: new evidence for compliance of the adaptation goals is continuously integrated to deal with the uncertainties that the system faces across its lifetime.
3. Reuse: the approach offers (i) a set of application independent templates for the design and verification of MAPE models; these templates can be instantiated for the domain at hand, and (ii) a reusable virtual machine that supports the execution of the feedback loop model at runtime.

Fig. 6.4 gives a high-level overview of ActivFORMS that spans four main stages of the software lifecycle of feedback loops: design, deployment, runtime adaptation, and evolution.

In Stage 1, *model and verify feedback loop*, based on stakeholder input, formal models for the feedback loop of the self-adaptive system are specified and verified.[1] In Stage 2, *deploy feedback loop model with virtual machine* the feedback loop model together with the ActivFORMS runtime environment are deployed. In Stage 3, *verify adaptation goals and adapt*, the feedback loop monitors the executing managed system and its environment and adapts the system to ensure the

[5] The total sum of the distribution factors for one mote is normally 100. If packets are duplicated and sent to more than one parent, the sum of the distribution factors will be above 100.

[1] We use *feedback loop model* to refer to the integrated model of the complete feedback loop, and *MAPE models* or *models of the feedback loop* to refer to the specific models of the different MAPE elements.

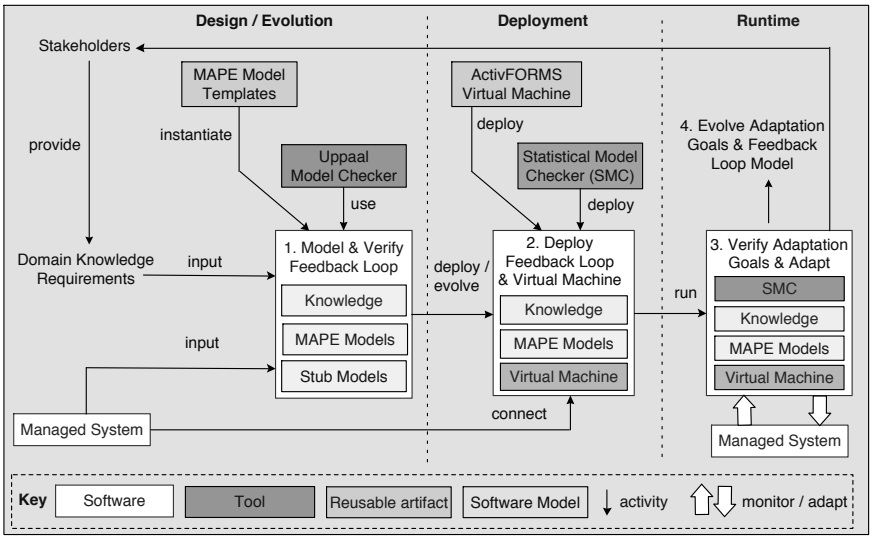


Figure 6.4: Overview of ActivFORMS approach

adaptation goals. In Stage 4, *evolve adaptation goals and feedback loop model*, the adaption goals and the deployed feedback loop model can be updated on the fly, for example to improve functions of the MAPE models or update the models to deal with a new or changing adaption goals.

We explain the four stages of ActivFORMS now in detail. We illustrate each phase with the DeltaIoT application. All reusable artifacts and concrete instantiations for DeltaIoT are available at the AvtivFORMS project website.[2]

6.4.1 Stage 1: Model and Verify Feedback Loop

The goal of the first stage of ActivFORMS is to develop formally verified models for the feedback loop of the self-adaptive system (*MAPE Models* and *Knowledge*, see Fig. 6.4). The input elements of Stage 1 are: (1) the managed system that is subject to adaptation; (2) domain knowledge; (3) a set of adaptation requirements; (4) a set of MAPE model templates; and (5) the Uppaal model checker.

Stage 1 consists of three main activities: *design feedback loop model*, *design stub models*, and *specify and verify properties*. We discuss these activities now in detail.

6.4.1.1 Design Feedback Loop Model

In the first activity, the formal models of the MAPE loop and the knowledge are specified (see Fig. 6.4). We start with the specification of the knowledge. Then we zoom in on the MAPE models.

Knowledge. The knowledge consists of elements that are shared among the MAPE elements. A central element of the knowledge is a *configuration* that captures the essential aspects of the managed system, its environment, and the qual-

[2] https://people.cs.kuleuven.be/ danny.weyns/software/ActivFORMS/

ities that are subject to adaptation. ActivFORMS provides a generic template to define configurations. Listing 6.2 shows the template to define configurations.

Listing 6.2: Template to define configurations.

```
type struct {
   <Element element>;
} ManagedSystem
type struct {
   <int qualityProperty>;
} Qualities
type struct {
   <int environmentProperty>;
} Environment
type struct {
   ManagedSystem <managedSystem>;
   Qualities <qualities>;
   Environment <environment>;
} Configuration
```

A *Configuration* is defined by the relevant elements of the managed system, a set of quality properties, and the relevant properties of the environment in which the managed system operates. To specialise the template for a concrete self-adaptive system, software engineers need to replace the generic elements marked between angle brackets with concrete domain specific elements. Additional elements may be introduced as needed.

Example 1. We illustrate now configurations for DeltaIoT, see Listing 6.3. The managed system of DeltaIoT consists of a set of motes. Each mote has a unique identifier, an energy level, and a set of links. Links are defined by a source and a destination mote, a power setting that is used by the source to send messages to the destination, and a distribution factor that determines the percentage of messages sent by the source over the link. The basic qualities of DeltaIoT configurations are packet loss and energy consumption. The environment of Delta IoT is characterised by two properties that represent uncertainties: the SNR per link and the traffic load generated per mote.

Listing 6.3: Specification of DeltaIoT configurations.

```
type struct {
   int sourceID; int destinationID; int powerSetting; int distributionFactor;
} Link
type struct {
   int moteId; int energyLevel; Link links[MAX_LINKS];
} Mote
type struct {
   Mote motes[MAX_MOTES];
} ManagedSystem
type struct {
   int packetLoss; int energyConsumption;
} Qualities
type struct {
   Load motesLoad[MAX_MOTES]; SNR linksSNR[MA_LINKS];
} Environment
type struct {
   ManagedSystem deltaIoT;
   Qualities qualities;
   Environment environment;
} Configuration
```

ActivFORMS provides a template for the specification of the knowledge; Listing 6.4 shows an excerpt of the template with the main elements.

Listing 6.4: Definition of Knowledge

```
//Knowledge =
//   {Configuration, Adaptation Goals, Adaptation Options, Plan, Quality
      Models}

<Configuration currentConfiguration>;

//Adaptation Goals
<bool optimizationGoal(Configuration gConf, Configuration tConf) {
  //Tests whether a test configuration (tConf) is more optimal regarding
  //a property as a given configuration (gConf) }>
<int PROP;>
<bool satisfactionGoal(Configuration conf, int PROP) {
  //Tests whether a configuration (conf) satisfies a given property (PROP) }>

//Adaptation Options
type struct {
  <ManagedSystem option>;
  <Qualities verificationResults>;
} AdaptationOption
AdaptationOption adaptationOptions[MAX_OPTIONS];

//Plan
type struct {
  <int stepType>;
  <Element element>;
  <int newValue>;
} <Step>
type struct {
  <Step steps[MAX_STEPS]>;
} Plan

//Quality Models
//A network of stochastic timed automata per quality model
```

Knowledge comprises five elements: the current *Configuration*, a set of *Adaptation Goals*, a set of *Adaptation Options*, i.e. the possible configurations of the managed system, a *Plan* consisting of adaptation steps that is dynamically composed by the *Planner* (the MAPE models are explained below), and a set of *Quality Models*, one model for each adaptation goal.

The configuration represents current knowledge of the feedback loop about the state of the managed system, the environment, and the qualities that are subject to adaptation, as defined in Listing 6.2.

The adaptation goals define the objectives that need to be realised by the feedback loop. In this paper, we represent adaptation goals as functions. We distinguish between an *optimizationGoal* that tests whether a configuration *tConf* is more optimal regarding a property as a given configuration *gConf*, and a *satisfactionGoal* that tests whether a configuration (*conf*) satisfies a given property (*PROP*). However, other types of adaptation goals can be defined and applied in ActivFORMS.

An adaptation option consist of two parts: a particular setting of the managed system (*option*) and a placeholder for the verification results (*verificationResults*). The *Analyser* dynamically computes the adaptation options. The verification results are added after the verifier has produced estimated values for the different

qualities per adaptation option. The *Planner* then selects the best option based on the verification results using the adaptation goals. In this paper, we assume that a limited but possible large number of adaptation options can be identified whenever adaptation is required. This implies that any system parameter that can be used for adapting the managed system with a value in a continuous domain needs to be limited in range and be discretised. Heuristics can be applied to select the adaptation options from a very large set, but this is out of scope of this paper.

A plan consists of a series of steps (*Step*), each defined by a *stepType*, the *element* of the managed system to which the step applies, and the *newValue* that needs to be applied to the element.

Finally, each quality model, specified as a stochastic timed automaton (or a network of these), captures the characteristics of one of the qualities that is subject of adaptation. A quality model, which is domain specific, is in essence an abstraction of the managed system and its environment comprising behaviour and state related to a particular quality property. This model enables a verifier to estimate the quality for a particular adaptation options. There are two types of parameters in the quality models: (1) parameters that correspond to settings of the managed system that are used to define adaptation options, and (2) uncertainties of the managed system and its environment.

Example 2. We illustrate the knowledge specification for DeltaIoT. Listing 6.5 shows an excerpt of the knowledge definition for a DeltaIoT feedback loop.

Listing 6.5: Knowledge definition for DeltaIoT feedback loop.

```
Configuration currentDeltaIoTConfiguration; //see Listing 3

//Adaptation Goals
int MAX_PACKET_LOSS = 10; //max packet loss 10%
bool satisfactionGoalPacketLoss(Configuration gConf, int MAX_PACKET_LOSS) {
    return gConf.qualities.packetLoss < MAX_PACKET_LOSS;
}
bool optimizationGoalEnergyConsumption(Configuration gConf, Configuration
     tConf) {
    return tConf.qualities.energyConsumption < gConf.qualities.
         energyConsumption;
}

//Adaptation Options
ManagedSystem deltaIoT_1 {...};
Qualities verificResults_1 = {...};  ...
AdaptationOption adaptationOptions[MAX_OPTIONS] =
   {{deltaIoT_1, verificResults_1}, {deltaIoT_2, verificResults_2}, ...};

//Plan with Step Types
const CHANGE_POWER;
const CHANGE_DISTRIBUTION;
Step step_1 = {CHANGE_POWER, link2_1, 5};
Step step_2 = {CHANGE_DISTRIBUTION, link2_1, 60};  ...
Plan plan  = {step_1, step_2, ... }
```

The current configuration of the DeltaIoT network (*currentDeltaIoTConfiguration*) is defined by the network of motes with the actual settings (transmission power settings of the motes, distributions of messages to parents), the current values of quality properties (power loss and energy consumption), and uncertainties (current traffic load of motes and SNR of links).

The basic setting of the system uses two adaption goals. The first goal, *satisfactionGoalPacketLoss*, tests whether the packet loss of a test configuration is not higher as a given threshold (here defined as 10%). The second goal, *optimizationGoalEnergyCons*, tests whether the energy consumption of a test configuration is lower as that of a given configuration.

The adaptation options are defined by the settings that can be used to adapt the network, i.e., transmission power and distribution of messages. The verification results are the estimated values for packet loss and energy consumption for the different settings of the managed system.

Two types of steps can be used to compose adaption plans: changing the power settings of a mote to transmit messages to the a parent, e.g., {*CHANGE_POWER, link2_1, 5*} says that the transmission power of mote 2 along the link to mote 1 is set to 5; and change the distribution of messages sent to parents, e.g., {*CHANGE_DISTR, link2_1, 60*} says that mote 2 will send 60% of its traffic to mote 1.

The knowledge of the basic feedback loop model for DeltaIoT comprises two quality models, one for packet loss and one for energy consumption. While these models have to be designed and deployed with the MAPE models, they are only used at runtime to support analysis. We discuss therefore the quality models in Stage 3, see Section 6.4.3.

MAPE Models. ActivFORMS provides generic model templates that support the specification of the MAPE feedback loops as a network of timed automata. These templates are derived from extensive experience with modelling MAPE loops for various applications, see for example [40, 94, 109, 170, 204, 209]. The templates target self-adaptive systems with characteristics as described in the introduction of this paper. The templates refine and significantly extend an initial set of model templates presented in [79].

Fig. 6.5 shows model templates to specify MAPE loop models. The templates use event triggering, e.g., the monitor automaton is activated by a probe signal with new data from the managed system and/or the environment. ActivFORMS also provide a set of time triggered templates, where MAPE models can be activated periodically by an internal clock (see the project website).

The *Monitor* waits in *Waiting* until it receives the *monitor* signal from the probe (the interaction with the probe is specified below). After initialising local variables (*initialize()*), the monitor updates the knowledge elements with the data collected by the probe, i.e., <*updateSystemData()*>, <*updateQualitiesData()*>, and <*updateEnvironmentData()*> respectively. Subsequently, the monitor checks *analysisRequired()* using the updated knowledge and triggers the analyser if analysis is needed (*analyze!*). The criteria to decide whether analysis is needed are domain specific (for example based on the degree of changes in the environment or the qualities of the system). If no adaptation is needed, the adaptation loop ends (*feedbackLoopCompleted!*). Before returning to the waiting state, the monitor may do some *postProcessing()*, e.g. resetting local variables.

When the *Analyser* is triggered, it starts with initialisation and then performs an analyses (*analyze()*) to check whether an adaptation is needed. The criteria to decide about the need for an adaptation are domain specific; they can be based on

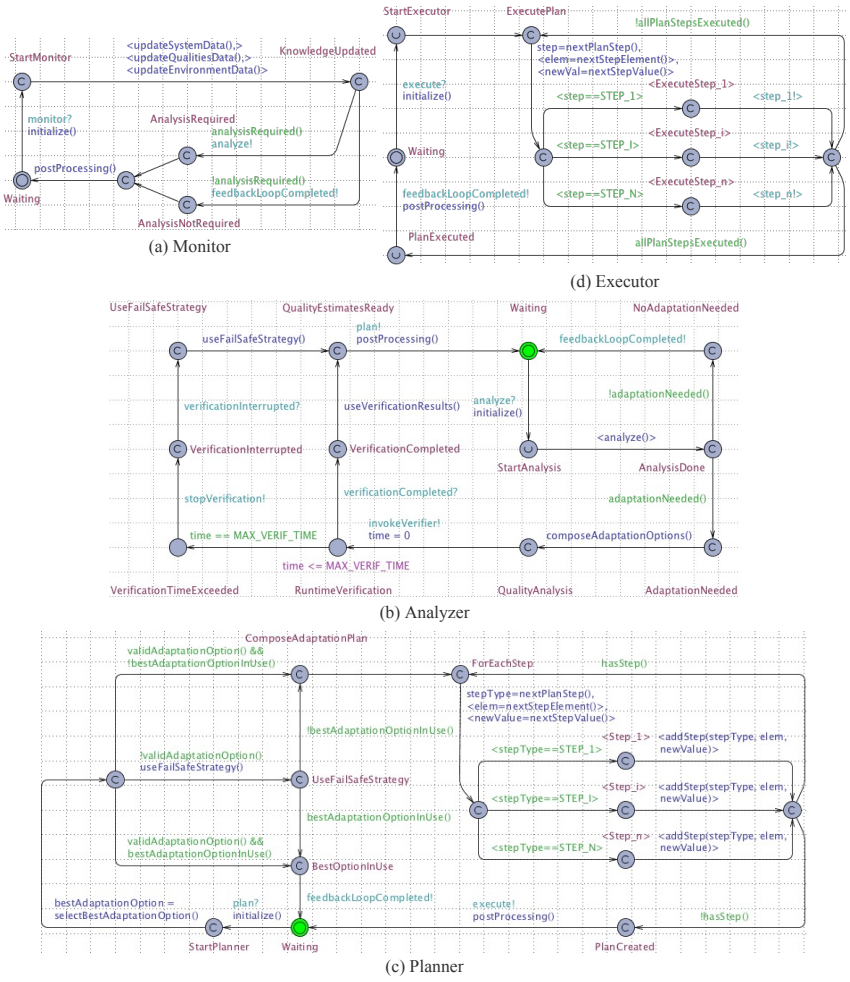


Figure 6.5: Reusable templates for specifying MAPE models

the degree to which properties monitored in the environment have been changed, the degree of changes of qualities of the managed system, etc. If no adaption is required, the feedback loop cycle ends (*feedbackLoopCompleted!*). If adaptation is required, the analyser composes the adaptation options. Each adaptation option is defined by assigning particular values to parameters of the managed system that allow adapting the system. For each adaptation option, placeholders are provided for the estimated quality properties. Subsequently, the analyser invokes the verifier via the *invokeVerifier* signal. The verifier uses the adaptation options and the quality models to compute for each adaptation option estimated values for the different qualities that are subject of adaptation. When the verifier completes verification (*verificationCompleted?*) it returns the verification results to the analyser. The analyser then applies *useVerificationResults()*, adding the verification results to the

placeholders for each adaptation option. However, if verification exceeds a max period of time (*MAX_VERIF_TIME*), the verification process is terminated. The verifier then returns partial results and the analyser applies *useFailSafeStrategy()*. The failsafe strategy can either use the partial results to update the adaptation options or it can decide to use a failsafe configuration as single adaptation option. Before triggering the planner, the analyser may do some *postProcessing()*.

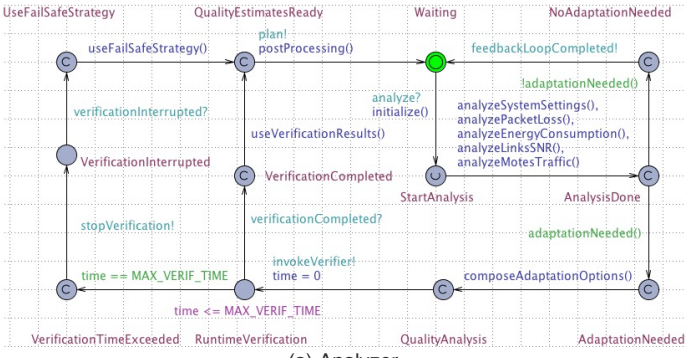
After initialisation, the *Planner* starts with selecting the *bestAdaptationOption* based on the quality estimates using the function *selectBestAdaptationOption()*. This function applies the adaptation goals one by one to the adaptation options. Now there are three possibilities. First: a *validAdaptationOption()* is found that is not in use (*!bestAdaptationOptionInUse*). In this case the planner will start composing an adaptation plan. Second: no valid adaptation option is found. In this case the failsafe strategy will be applied. In case the resulting configuration is not in use, the planner will start composing an adaptation plan; otherwise planning will complete (*feedbackLoopCompleted!*). Third: the best adaptation option is already in use. This will immediately complete planning. Composing a plan is done step by step. For each step, the *stepType* is determined, the element (*elem*) of the managed system that is subject of the adaptation step is identified, and the *newValue* that needs to be used to adapt the element is determined. These data elements are directly derived from the adaptation option that is selected for adaptation; only for the elements of the managed system that are changed a plan step will be generated. The step is then added to the plan (*addStep(stepType, elem, newValue)*). When all the required steps are added to the plan (*!hasStep*) the executor is triggered via the *execute* signal, possibly involving some *postProcessing*.

Finally, the *Executor* executes the plan when triggered by the planner. After initialisation, the plan is executed step by step (*ExecuteStep_i*). The signal *step_i!* triggers the effector to perform the adaptation action to the element of the managed system defined by that step. When the plan is completed (*allPlanStepsExecuted*), the executor can perform some final *postProcessing* completing the feedback loop cycle.

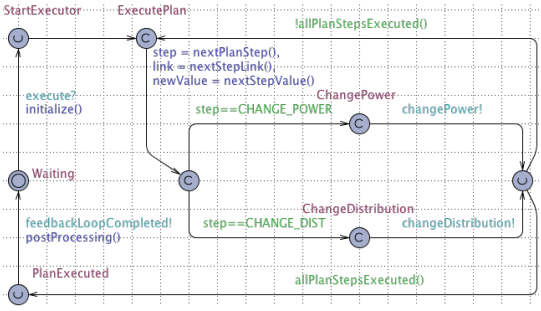
To specialise the model templates for a given system, software engineers need to: (1) replace the generic signals and functions marked between angle brackets with concrete domain specific instances, and (2) implement the abstract functions and guards for the domain at hand. In addition, particular elements of the models can be refined when needed.

Example 3. Fig. 6.6 shows concrete instances of the templates for the analyser and executor models for DeltaIoT. The instantiations for the monitor and planner are available at the project website.

The *Analyzer* uses a set of domain-specific functions to analyse the current setting. *analyzeSystemSettings()* checks whether the network settings (power and distribution per link) are different from the expected settings (as applied in the last adaptation step). A difference indicates that the last adaptation steps were not effected as expected or the settings changed for another reason. *analyzePacketLoss()* and *analyzeEnergyConsumption()* check whether the packet loss and energy consumption have increased significantly. Similarly, *analyzeLinksSNR()* and *analyze-*



(a) Analyzer



(b) Executor

Figure 6.6: MAPE feedback loop model for DeltaIoT

MotesTraffic() check the changes of *SNR* of the links and the traffic load generated by the motes. If any of the analysis functions return true, adaptation is needed (*adaptationNeeded()*). The adaption options are then composed in two steps.

In the first step, the analyser computes the power setting per link that ensures reliable communication (i.e. *SNR* equal or just above zero). To that end, the analyser uses link-specific functions $SNR = \alpha + \beta(power)$, where α and β are values that are determined per link before deployment based on a series of experiments. Fig. 6.7 shows an example of a function for one of the links of the DeltaIoT network. The analyser determines the required power setting per mote and per link iteratively. The analyser starts with computing the *SNR* for the current power setting using the function $SNR = \alpha + \beta(power)$ for the link. Then the analyser compares the computed *SNR* with the current *SNR* measured by the probe; we denote the difference as the *SNR delta*. If the computed value minus the delta is lower than zero the power setting is incremented and the *SNR delta* is computed again. This process is repeated until an *SNR* of at least zero is found. If the computed value minus the delta is above zero the reduced power setting is reduced and the process is repeated until an *SNR* of zero or just above zero is found. These power settings are then used for all adaptation options.

In the second step, all the possible combinations of message distributions for all links to all its parents are determined (in a range of $[0...100]$ in steps of 20) (if there

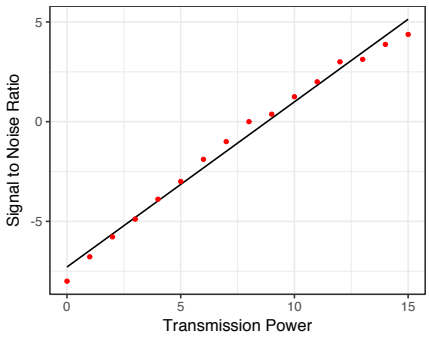


Figure 6.7: SNR to Power for one of the links of the DeltaIoT network ($\alpha = -7.29$ and $\beta = 0.83$)

is only one parent, all messages are sent to that parent). Each of these combinations determine an adaptation option. Under the assumption that the network structure does not change, these combinations do not change.

The *MAX_VERIF_TIME* in the DeltaIoT configuration is set for the given configuration (e.g., for a setting with a cycle time of 9.5 minutes, the max verification time is set is to 8 minutes). The function *useVerificationResults()* copies the estimated values for packet loss and energy consumption determined by the verifier for all adaptation options.

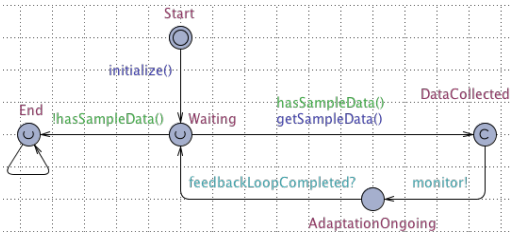
In DeltaIoT, we use the following failsafe strategy: if the partial verification results contain at least one adaptation option that satisfies the adaptation goals the best option is selected among these; if there is no such option, the settings of the reference approach are applied with maximum power settings for each mote and duplication of messages send to all parents.

The *Executor* of DeltaIoT applies two types of steps: *ChangePower* that adapts the transmission power of a *link* with a *newValue*, and *ChangeDistribution* that adapts the percentage of messages distributed of a link with a new value.

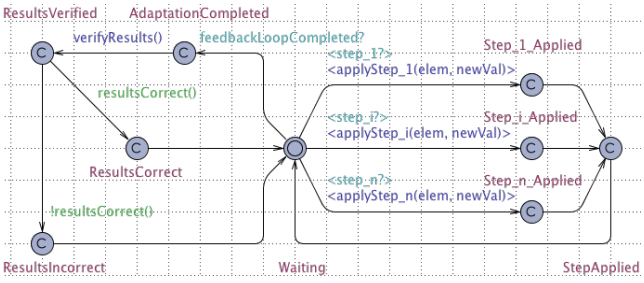
6.4.1.2 Design Stub Models

Recall that ActivFORMS assumes that the managed system is available, as a legacy or a greenfield system, and that it is equipped with the necessary probes and effectors. The verification of feedback loop models requires stub models (or stubs) that capture the essential behaviours of the managed system, the environment, together with probes and effectors. In addition, a stub model is required that captures the essential behaviour of the runtime verifier.

ActivFORMS provides a set of generic templates to define these stubs that need to be instantiated for the given adaptation problem. To enable the verification of the correctness of the feedback loop (see below), the stubs have to exercise the possible paths through the MAPE models. This requires that the stubs take the necessary input for the domain at hand, ensuring that all the different guard combinations are satisfied.



(a) Probe stub



(b) Effector stub

Figure 6.8: Reusable templates for probe and effector stubs

Fig. 6.8(a) shows the *Probe stub*. From the *Start* state, the function *initialise()* enables to configure the setting. The probe then starts collecting sample data from the system, the relevant qualities, and the environment (*getSampleData()*). The sample data is typically specified as a sequence of configurations (see Listing 6.2). Once the data is collected, the probe triggers the monitor model of the feedback loop via the *monitor* signal, which starts an adaptation cycle. As soon as the feedback loop cycle completes (*feedbackLoopCompleted* signal), the probe enters the *Waiting* state from where it starts a new cycle as long as sample data is available (*hasSampleData()*). The probe completes (*End*) when all sample data is processed.

Fig. 6.8(b) shows the *Effector stub*. In the *Waiting* state, the effector receives the adaptation steps from the executor model. When a step is received, e.g., *<step_i?>*, the effector applies the adaptation action on the corresponding element of the managed system with the new value (i.e., *<applyStep_i(elem,newVal)>*). Once the adaptation plan is executed, the feedback loop notifies the effector via the *feedbackLoopCompleted* signal. The effector can then check the adaptation results. The *verifyResults()* function checks whether the configuration is adapted as expected or not (*ResultsCorrect* or *ResultsInCorrect*). The effector then returns to the *Waiting* state.

Fig. 6.9 shows the template for the *Verifier stub*. When the stub receives the *invokeVerifier* signal it collects a sample with estimates of the qualities for a set of adaptation options (predefined by the engineer). This set contains either complete or partial verification results. In case the set is complete, control is returned (*verificationCompleted!*) to the analyser that can use the verification results. In case the set is incomplete, the stub waits in *PartiallyVerified* until the analyser times out (verification exceeds *MAX_VERIF_TIME*). The analyser is then notified that

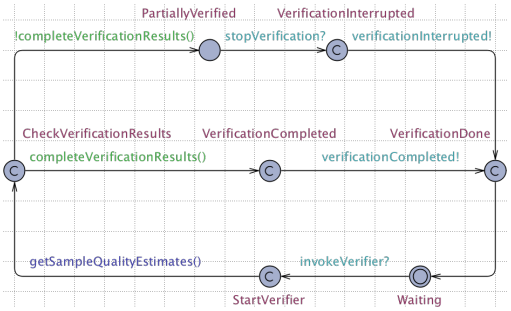


Figure 6.9: Template for the verifier stub

the verification was interrupted (*verificationInterrupted!*), after which the analyser uses the failsafe strategy.

Example 4. Fig. 6.10 illustrates how the effector stub is instantiated for DeltaIoT. The other stub instances are available at the project website.

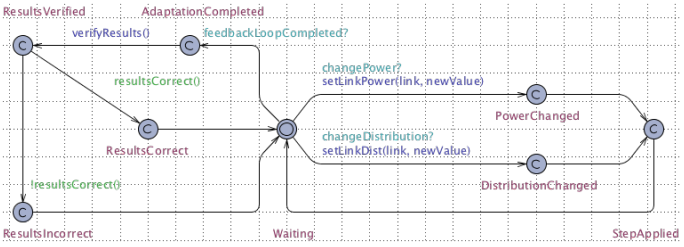


Figure 6.10: Effector stub for DeltaIoT

The effector stub distinguishes between two types of steps: *changePower?* and *changeDistribution?* The former one sets the transmission power for a given *link* to *newValue*; the latter sets the the distribution factor for a given *link* to *newValue*.

6.4.1.3 Specify and Verify Properties

In the third activity of Stage 1, we specify and verify properties that check the correctness of MAPE models. We specify properties in Timed Computation Tree Logic (TCTL). As explained in Section 6.2, TCTL expressions allow to verify properties such as safety, liveness, etc. We use the Uppaal tool [16] to verify the MAPE models.

ActivFORMS provides a set of reusable properties that all MAPE models specified with the behaviour templates should comply to. This set of properties refine and significantly extend an initial set of properties defined in [79]:

```
Pr1. Probe.DataCollected --> Monitor.KnowledgeUpdated
Pr2. Monitor.AnalysisRequired --> Analyzer.AnalysisDone
Pr3. Analyzer.AdaptationNeeded --> Verifier.VerificationDone
Pr4. Analyzer.QualityEstimatesReady -->
     Planner.ComposeAdaptationPlan || Planner.BestOptionInUse
Pr5. Planner.<Step_i> --> Executor.<ExecuteStep_i>
Pr6. Executor.<ExecuteStep_i> --> Effector.<Step_i_Applied>
```

```
Pr7. Planner.PlanCreated --> Executor.PlanExecuted
Pr8. Executor.PlanExecuted --> Effector.AdaptationCompleted
Pr9. Analyzer.VerificationTimeExceeded -->
     Analyzer.UseFailSafeStrategy
Pr10 A[] !Effector.ResultsIncorrect
Pr11 E<> <Model.Location>
Pr12. A[] no deadlock
```

Property $Pr1$ states that when the probe collects data, the monitor will eventually use this data to update the knowledge in the repository. Property $Pr2$ states that when the monitor identifies the need for an analysis, eventually the analyzer will perform the analysis. Property $Pr3$ states that when the analyzer determines the need for an adaptation, eventually the verifier will perform the verification. Property $Pr4$ states that when the quality estimates of the adaptation options are ready, eventually the planner will either compose an adaptation plan in case the current configuration needs to be adapted or no new plan is required in case the best option is already in use. Properties $Pr5$ and $Pr6$ state that each step of the planner is eventually applied by the effector. Properties $Pr7$ and $Pr8$ state that when the planner has created a plan, eventually the plan is executed by the effector via the executor. Property $Pr9$ states that when runtime verification exceeds a predefined maximum time (*MAX_VERIF_TIME*), the analyser will use a failsafe strategy to adapt the system. Property $Pr10$ states that location *ResultsIncorrect* of the *Effector* model is never reached. Property $Pr11$ on the other hand states that there exists a path to a given *Location* of a given *Model*; both location and model are abstractly defined. For example, *Planner.UseFailSafeStrategy* checks that a path exists to location *UseFailSafeStrategy* of the *Planner* model. This abstract property allows checking whether the input used for verification is complete, i.e. all paths of the models are traversed. Property $Pr12$ that is supported by Uppaal allows verifying whether the system is deadlock free. The elements in angle brackets need to be replaced by the domain specific instances of the concrete MAPE models.

Example 5. We illustrate the property specification and verification for DeltaIoT. Properties $Pr1$ to $Pr4$ and $Pr7$ to $Pr10$ and $Pr12$ can be directly applied to the MAPE feedback loop model for DeltaIoT (see Figure 6.6). $Pr5$ and $Pr6$ and $Pr11$ need to be instantiated for DeltaIoT:

```
//Generic property:
Pr5. Planner.<Step_i> --> Executor.<ExecuteStep_i>
Pr5a. Planner.ChangePower --> Executor.ChangePower
Pr5b. Planner.ChangeDistribution --> Executor.ChangeDistribution

//Generic property:
Pr6. Executor.<ExecuteStep_i> --> Effector.<Step_i_Applied>
Pr6a. Executor.ChangePower --> Effector.ChangePower
Pr6b. Executor.ChangeDistribution --> Effector.ChangeDistribution

//Generic property: Pr11 E<> <Model.Location>
Pr11. E<> Planner.UseFailSafeStrategy (selected property)
```

Figure 6.11 shows the time taken to verify the MAPE feedback loop properties for a setup of DeltaIoT with 15 motes as shown in Figure 6.11 (averages of 30

runs). For these experiments, we used a feedback loop that deals with two adaptation requirements (R1. Packet loss should be less than 10% and R2. Energy consumption should be minimised). The models and stub data that we used for the verification with the Uppaal model checker are available at the project website. The results show that the overhead for design-time verification of the properties that check the correctness of the MAPE feedback loop is totally acceptable for this realistic IoT setting.

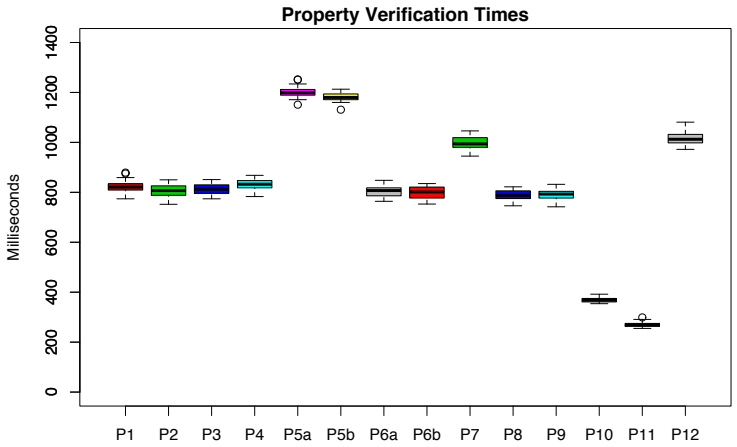


Figure 6.11: Verification times for properties that check the correctness of the MAPE feedback loop

6.4.2 Stage 2: Deploy Feedback Loop with Virtual Machine

The goal of the second stage is to deploy and enact the verified feedback loop model with the ActivFORMS virtual machine. The input elements of Stage 2 are: (1) the verified MAPE-K feedback loop model; (2) the ActivFORMS virtual machine; (3) a statistical model checker; and (4) template classes for connecting the probe, effector, and statistical model checker with the adaptation logic.

Stage 2 consists of four main engineering activities: *instantiate virtual machine with feedback loop model*, *connect probe and effector*, *connect statistical model checker*, and *start virtual machine*. We briefly explain each activity now.

6.4.2.1 Instantiate Virtual Machine with Feedback Loop Model

One of the distinct features of ActivFORMS is a virtual machine that can directly execute the verified feedback loop model to adapt the managed system. Direct model execution avoids model to code translation which is typically an error-prone activity. We start with a brief overview of the ActivFORMS virtual machine. Then we show how the virtual machine is instantiated with the feedback loop model of the DeltaIoT application. For a detailed explanation of the virtual machine, we refer to [107].

The virtual machine can interpret the feedback loop model according to the semantics of networked timed automata. Internally, the virtual machine uses stan-

dard compiler techniques to transform the models of the feedback loop with their locations and edges to an internal graph representation. The labels on the edges and states, e.g., guards, invariants, etc. are converted to task graphs. A task graph consists of a list of tasks that need to be executed when activated, such as updating a variable, evaluating an expression, etc. Once the model is converted to a graph representation, the virtual machine initialises all the signals and assigns a unique identifier to each signal. The feedback loop can then be connected with external components using the signal identifiers. ActivFORMS provides a set of template classes to connect probes, effectors and a verification engine with the feedback loop model. We explain the connection of these elements further in the following sections. Once all external connections are estabilised and the models are initialised, the feedback loop model is ready to be executed by the virtual machine.

In addition to the set of predefined classes that enable the feedback loop models to communicate with external components, the virtual machine also offers a generic plug-in mechanism to attach arbitrary components to the virtual machine. A plug-in that comes with ActivFORMS is a graphical user interface that visualises the executing model and shows the status of user defined properties. Another plug-in provided by ActivFORMS is a live update module that allows to update running models on the fly. We elaborate on these two plug-ins in Stage 3 (verify adaptation goals and adapt the managed system) and Stage 4 (evolution of models and adaptation goals).

The ActivFORMS virtual machine has been thoroughly tested using an exhaustive test suite. Furthermore, the virtual machine has been applied in a wide range of self-adaptive systems, including a robotic system [109], in several case studies including a smart house system, a security system, and two vehicular traffic systems [94], a digital story telling application [94], an e-health system [170, 206], an unmanned underwater vehicle system and an service-based foreign exchange trading system [40]. More information about how the virtual machine works and the test suite with test results is available at the project website.

Example 6. Listing 6.6 shows how an instance of the ActivFORMS engine is created, taking as input the path where the MAPE feedback model of DeltaIoT is located.

Listing 6.6: Instantiate ActivFORMS with feedback loop of DeltaIoT.

```
ActivFORMSEngine engine = new ActivFORMSEngine("/models/DeltaIoT-MAPE.xml");
```

6.4.2.2 Connect Probe and Effector

Recall that ActivFORMS assumes that the managed system is equipped with probes and effectors. To enable the MAPE feedback loop to monitor the managed system and the environment through these probes and adapt the managed system with the effectors, the feedback loop needs to be connected to the probe and the effector. To that end, ActivFORMS provides two template classes, *ProbeConnector* and *EffectorConnector*, that support the engineer to implement these connections. Realising the probe connection boils down to: (1) connect the monitor model with the probe via the monitor channel, (2) implement the logic to receive data from

the probe, (3) translate the data in a format that the monitor understands, (4) send the translated data to the monitor. Realising the effector connection consists of: (1) connect the executor model with the effector via the respective channels; (2) effect the adaptation actions received from the executor model via the channels to the managed system via the effector.

Example 7. Listing 6.7 shows how a connector is created that connects the DeltaIoT probe with the monitor model through the monitor channel and how it sends data from the probe to the monitor.

Listing 6.7: Connecting DeltaIoT probe with the monitor model.

```
public ProbeConnector(ActivFORMS engine, Probe probe) {
  // Connect probe with monitor via the monitor channel identifier
  monitor = engine.getChannel("monitor");
}
public void sendDataToFeedbackLoop() {
  // Get data from probe
  List<Mote> motes = probe.getAllMotes();
  // Convert data into ActivFORMS readable format
  data[0]="configuration.deltaIoTmotes[2].energyLevel="
               + motes.get(2).energyLevel;
  data[1]="configuration.environment.linksSNR[0].SNR="
               + motes.get(2).links.get(0).SNR;
  ...
  // Send data from probe to monitor
  engine.send(monitor, data);
}
```

Listing 6.8 shows how a connector is created that connects the executor model with the effector and how the effector receives adaptation actions from the executor to effect the network settings.

Listing 6.8: Connecting the executor model with the DeltaIoT effector.

```
public EffectorConnector(ActivFORMSEngine engine, Effector effector) {
   // Get channel identifiers from engine
  changePower = engine.getChannel("changePower");
  changeDistribution = engine.getChannel("changeDistribution");
  // Connect executor model with effector via channels
  engine.register(changePower, "link", "newVal");
  engine.register(changeDistribution, "link", "newVal");
}
@Override
public synchronized void receive(int channelId, HashMap data) {
  if (channelID == changePower){
     // effect power settings for link through effector
  }
  else if (channelId == changeDistribution) {
     // effect distribution settings for link through effector
  }
}
```

6.4.2.3 Connect Statistical Model Checker

In ActivFORMS we use statistical model checking at runtime to support the analysis of adaptation options, i.e., to estimate the expected quality properties of the adaptation goals. To enable the analyzer model to use the model checker the two need to be connected. ActivFORMS provides a template class, *SMCConnector*, that supports the engineer to implement this connection. Realising the connection consists of the following steps: (1) connect the necessary channels, including a

channel to invoke the verifier, a channel to stop verification (in case a time out is detected by the analyzer), and a channel to return results when verification is completed, (2) implement the logic to invoke the verifier for the different adaptation options and collect the results, (3) implement the logic to stop the verification process in case of a time out.

Example 8. Listing 6.9 shows how the analyser model for DeltaIoT is connected with the SMC to support the analysis of adaptation options at runtime.

Listing 6.9: Connecting the analyzer model of DeltaIoT with statistical model checker.

```
public SMCConnector(ActivFORMSEngine engine, SMC smc) {
  // Get channel identifiers and connect
  invokeVerifier = engine.getChannel("invokeVerifier");
  stopVerification = engine.getChannel("stopVerification");
  verificationCompleted = engine.getChannel("verificationCompleted");
  engine.register(...);
}
// Invoke verification and collect results
@Override
public synchronized void receive(int channelId, HashMap adaptationOptions) {
  if (channelID == invokeVerifier) {
     // For each adaptation option invoke SMC to estimate quality properties;
     // Once all are done send results through the done signal
  }
  else if (channelId == stopVerifier) {
     // Stop verification and send partial results through done signal.
  }
}
```

6.4.2.4 Start Virtual Machine

When the ActivFORMS engine is created using the feedback loop model as input and all the connections are established (with the probe, effector, and the verifier), one more aspect needs to be dealt with before the virtual machine can be started. In particular, the engineer needs to define the real time that corresponds to one logical time unit in the model. The virtual machine provides a function *setRealTimeUnit(int msTime)* to support the engineer with this setting. Once this is done the virtual machine can be started, enacting self-adaptation.

Example 9. Listing 6.10 shows the steps to start the virtual machine for DeltaIoT.

Listing 6.10: Start the virtual machine for the DeltaIoT network.

```
public void startAdaptation()){
   ActivFORMSEngine engine;
   engine = new ActivFORMSEngine("/models/DeltaIoT-MAPE.xml");
   // Set model time unit to real time unit in milliseconds
   engine.setRealTimeUnit(1000);
   // Initialize connections
   ...
   // Start the virtual machine
   engine.start();
}
```

An instance of the virtual machine is created with the path where the feedback loop model of DeltaIoT is situated. The real time that corresponds with one time tick on the model is set to 1000. When the external connections with the probe, effector, and the verifier are set, the engine is started.

6.4.3 Stage 3: Runtime Verification of Adaptation Goals and Decision Making

The goal of the third stage is to ensure the adaptation goals. To that end, the verified feedback loop that is executed by the ActivFORMS virtual machine adapts the managed system. The input elements of Stage 3 are: (1) the verified MAPE-K feedback loop model running on the virtual machine; (2) a statistical model checker connected with the analyzer model; and (4) the managed system connected with the monitor and executor models. Facultative, the GUI can be connected to the virtual machine allowing to inspect the running feedback loop model and user defined properties.

We focus in Stage 3 on three main activities: *analysis of adaptation options*, *decision making*, and *inspection of the feedback loop in execution*. Before we explain these three activities in detail, we first give an overview of the runtime architecture of ActivFORMS.

6.4.3.1 Runtime Architecture of ActivFORMS

The focus of ActivFORMS in Stages 1 and 2 (design and deployment time) is on providing guarantees for functional correctness. This concerns both correctness of the MAPE models and the virtual machine that executes the models. At runtime, ActivFORMS complements functional correctness with guarantees for the adaptation goals. ActivFORMS aims to provide these guarantees in an efficient way, i.e. with limited resources and adaptation time. To that end, ActivFORMS relies on statistical model checking at runtime that can provide guarantees for the adaptation goals with a required level of confidence. We give a high-level overview of the runtime architecture of ActivFORMS that shows the composition of the runtime components. In the following sections, we zoom in on analysis and decision making using runtime statistical model checking. Fig. 6.12 gives an overview of the ActivFORMS runtime architecture.

The *Managed System* is the software that is subject of adaptation. At a given time the managed system has a particular configuration that is based on the arrangement and settings of the running components that make up the system. The managed system is equipped with probes and effectors.

In line with the reference model of Kramer and Magee [122], the *Managing System* comprises two sub-layers: *Change Management* and *Goal Management*. Change management contains a feedback loop that is connected with the probes and effectors of the managed system. Change management adapts the managed system at runtime. Goal management on the other hand enables to inspect change management and update the components of this layer on-the-fly.

As we explained before, in ActivFORMS the feedback loop of change management is realised by means of a network of timed automata that are directly executed by the ActivFORMS virtual machine. In Fig. 6.12, these models are represented as MAPE components that share knowledge maintained in the *Knowledge Repository*. As the virtual machine executes the formally verified MAPE models according to the semantics of timed automata, correctness is guaranteed at runtime.

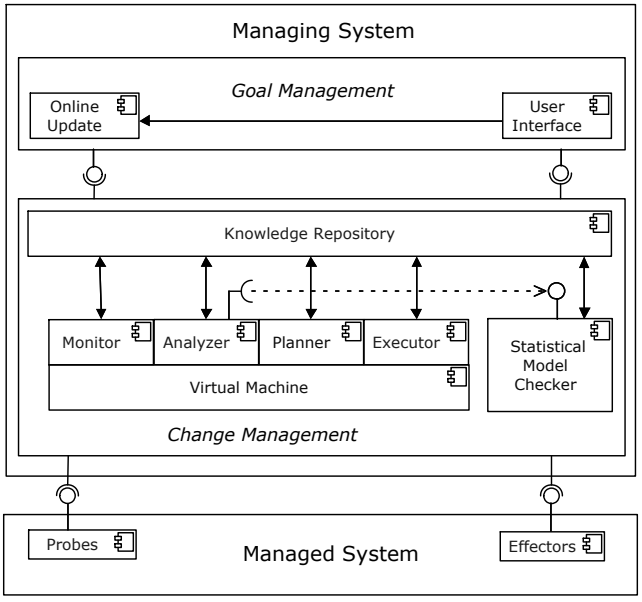


Figure 6.12: ActivFORMS runtime architecture

The *Monitor* uses *Probes* to monitor the managed system and the environment in which the system operates. The collected data can be used to update the knowledge of the knowledge repository directly, or learning mechanisms can be used, for example Bayesian learning can use new data that becomes available to update the probability of a parameter that quantifies an uncertainty of the system or the environment. The *Analyzer* is supported by a *Statistical Model Checker* that can run simulations on the models of the knowledge repository during operation. In particular, the model checker can provide estimates of the quality properties for each adaptation option of the system that is subject of adaptation. Recall that we refer to the different choices for adaptation from a given configuration of the managed system as the adaptation options. The *Planner* uses the adaptation goals to select the best adaptation option and create a plan to adapt the managed system accordingly. This plan is then executed by the *Executor* using the *Effectors*. In this paper, our primary focus is on analysis of the adaptation options and the selection of an option; we discuss these in detail below.

Goal Management consists of two complementary plug-in components, *User Interface* and *Online Update*. The user interface component allows visualising the executing feedback loop model and showing the state of user-defined properties. The online update component enables users to update MAPE models or elements of the knowledge repository during execution. Such updates are submitted through the user interface that forwards the request to online update. Changing the models and knowledge of change management needs to be done safely, i.e., in quiescent states [123]. We discuss goal management in detail in the next Section 6.4.4.

6.4.3.2 Analysis of Adaptation Options

The goal of analysis is to provide estimates for the different quality properties of the adaptation options. In ActivFORMS, analysis is performed on first-class runtime models for each quality property of interest using statistical model checking. Analysis consists of four steps: (1) compose the adaptation options by assigning values to the variables that represent the elements of the managed system that can be adapted, (2) assign values to the uncertainties as observed by the monitor and stored in the knowledge repository; the uncertainties are represented as variables in the model, (3) invoke the model checker with the adaptation options and the verification queries for the different quality models, (4) collect the verification results and use them to update the quality estimates of the adaptation options in the knowledge repository. In case the verification process exceeds a pre-defined time limit, verification is interrupted and a failsafe strategy is applied to decide about adaptation. This completes analysis.

We rely on Uppaal-SMC for statistical model checking at runtime using two types of queries: probability estimation ($p = Pr[bound](\varphi)$) and simulation ($simulate\ N[\leq bound]\{E1, ..., Ek\}$). For a probability estimation query the statistical model checker applies statistical techniques to compute the number of runs needed to produce a probability estimation p for expression φ of the quality model with an approximation interval $[p-\epsilon, p+\epsilon]$ and confidence $(1-\alpha)$ for a given time $bound$. The values of ϵ and α that determine the the accuracy of the results can be set for each query. For a simulation query, the value of N determines the number of simulations the model checker will apply in time $bound$ to return values for state expressions $E1, ..., Ek$ of the quality model. For this type of query, it is the responsibility of the designer to determine how many runs are needed to obtain a required accuracy. In our current research, we use the relative standard error of the mean (RSEM) as a measure to determine the accuracy of the simulation queries. The standard error of the mean (SEM) quantifies how precisely a simulation result represents the true mean of the population (and is thus expressed in units of the data). SEM takes into account the value of the standard deviation and the sample size. RSEM is the SEM divided by the sample mean and is expressed as a percentage. For example, a RSEM of 5% represents an accuracy with a SEM of plus/minus 0.5 for a mean value of 10. Evidently, better estimates require smaller RSEM values and thus more simulation runs. The number of simulations required for a particular accuracy can be determined during offline experiments. If necessary, additional experiments can be run in the background in case of significant model changes. In our current research, we rely on offline experiments only.

Example 10. We illustrate now the analysis of adaptation options for packet loss in DeltaIoT. For the analysis of energy consumption, we refer to the project website.

The quality model for packet loss consists of two interacting automata: $Topology$ and $Network$, shown in Fig. 6.13. The model is used to estimate the packet loss for a given adaptation option, using the following query:

Listing 6.11: Verification query for packet loss model

```
Pr [<=1](<>Network.PacketLoss)
```

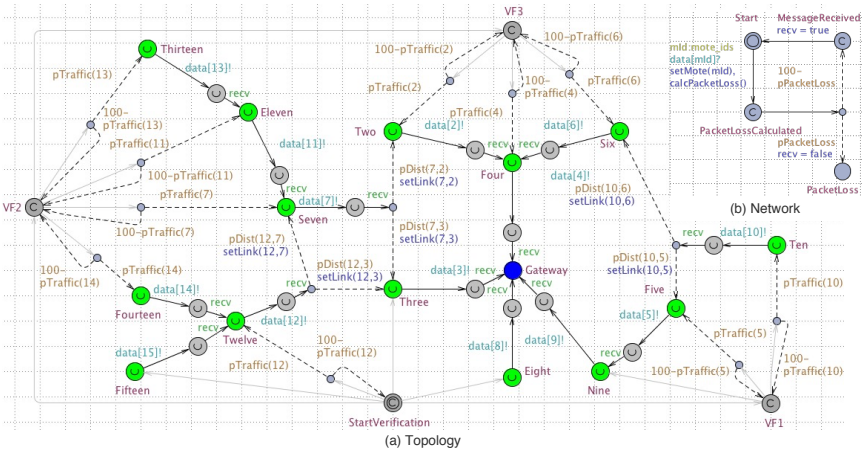


Figure 6.13: Quality model to estimate packet loss

We consider two types of uncertainties: the traffic load generated by the different motes and the signal to noise ratio per link (SNR). For these uncertainties we need to assign values in the model before the verification starts. A number of motes generate a steady traffic load (i.e., motes $Three$, $Eight$, $Nine$, and $Fifteen$; these mote periodically sample the temperature, see Fig. 6.2). The load generated by these motes is represented by constants. Other motes generate a fluctuating traffic load (i.e. based on the presence of humans). The load generated by these motes is represented probabilistically in the model; e.g., $pLoad(13)$ is the probability (expressed as a percentage) that mote $Thirteen$ will generate traffic. The values for $pLoad$ are periodically updated by the gateway and collected by the probe. The SNR per link depends on external factors such as network interference and noise in the environment. For each link, the values for SNR are periodically updated by the gateway and collected by the probe. These values are used to determine the values of SNR for the model as explained in Example 3. The values for traffic and SNR are assigned before verification starts. For a network with 15 motes as shown in Fig. 6.2, this results in 294 adaptation options.

The *Topology* automaton consists of 14 motes (*Two* to *Fifteen*) that send data to the *Gateway* via their parents (see also Fig. 6.2). As explained in Section 6.3, data communication in DeltaIoT is time synchronised, where communication over links is turn-based. *StartVerification* (or any of the connected locations *VF1*, *VF2* or *VF3*) triggers the mote that can communicate. The motes communicate such that the packet loss along all the possible paths in the network are checked.

If a mote with only one parent communicates (e.g. *Thirteen* communicates with *Eleven*), it signals the network automaton (*data[moteId]!*), where *moteId* is the ID of the sending mote (ranging from 2 to 15). The network automaton then determines the packet loss (see below). If a mote has multiple parents the distribution of data communicated to the parents is determined probabilistically based on the values assigned for the adaptation option. E.g., the probability that mote *Twelve*

sends data to mote *Seven* is *pDist(12,7)*, while the probability that it sends data to mote *Three* is *pDist(12,3)*.

We now look at the *Network* automaton of the packet loss model. When a mote (with identifier *mId*) communicates with a parent, the network automaton receives the *data[mId]?* signal with the mote that sends data (*setMote(mId)*). The probability for packet loss is then calculated using the $calcPacketLoss()$ function. The probability that packets get lost during communication depends on the Signal-to-Noise ratio (SNR) for the link. The values for the SNR along the different links are periodically updated by the gateway and collected by the probe and the monitor.

Depending on the value of the packet loss either the transition *PacketLossCalculated* to *PacketLoss* is taken (communication failed) or the transition to *MessageReceived* is taken (communication was successful). After a successful communication, the network automaton moves back to the *Start* location setting the value *recv=true*. The *Topology* automaton will then continue with the next hop of the communication along the path that is currently checked, until the *Gateway* is reached. If a packet gets lost (*recv=false*), the communication along the path that is currently checked ends. The verification process repeats until results with the required accuracy are obtained.

6.4.3.3 Decision Making

The goal of decision making is to pick the best adaptation option based on the analysis results and the adaptation goals. ActivFORMS does not prescribe what decision making mechanism should be used. Any approach to define adaptation goals and mechanism to select among the adaptation options based on the goals can be applied. ActivFORMS comes with two predefined types of adaptation goals that are defined as boolean functions: an optimization goal returns the most optimal configuration of two given configurations for a given property, and a satisfaction goal tests whether a given configuration satisfies a given property. The concrete goals are applied in a predefined order to determine the best adaptation option.

Example 11. We illustrate now the decision making applied in DeltaIoT for a setting with 15 motes and two adaptation goals. Packet loss is defined as a satisfaction goal (packet loss $< 10\%$) and energy consumption as a optimization goal (minimize energy consumption) For the definitions of the functions of the goals, we refer to Listing 6.5. First packet loss is applied, then energy consumption.

Figure 6.14 shows an overview of the adaptation options for DeltaIoT at a particular point in time. Each dot on the graph at the left hand side represents an adaption option with its corresponding average values of the two quality properties. The dot marked in blue represents the configuration of the managed system in use at the time the analyse is performed. The dots marked in green on the graph at the right hand side represent adaptation options that comply with the adaptation goal for packet loss. Finally, the dot marked in red on this graph represents the best adaptation option, i.e., the candidate option with minimum energy consumption. This option is selected for adaptation and a plan is composed by the planner that adapts the current configuration to this new configuration. The selected adaptation option at this particular point in time is expected to reduce packet loss to 9% and

energy consumption to 25.6 compared to 11% and 25.7, i.e., the respective values of the current configuration.

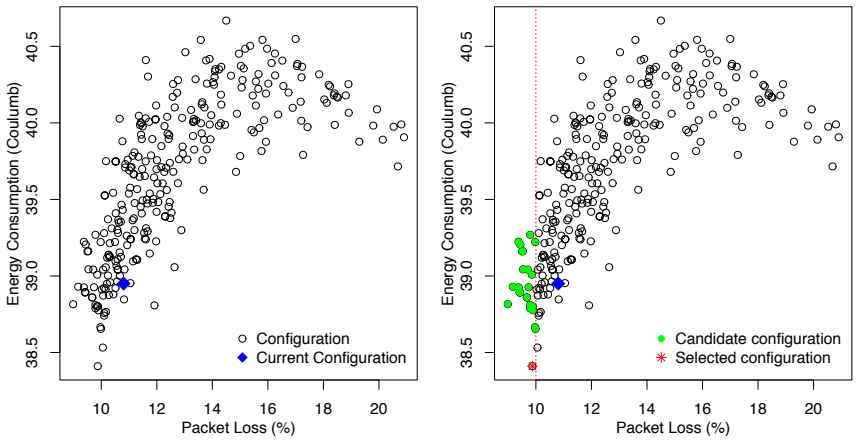


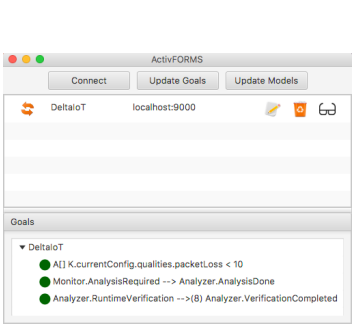
Figure 6.14: Decision making at a particular point in time with two adaptation goals

6.4.3.4 Inspect the Feedback Loop in Execution

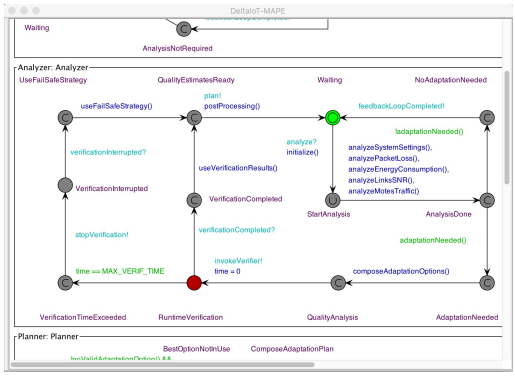
The third and last activity of Stage 3 is inspection of the executing feedback loop model. The GUI plugin that comes with ActivFORMS allows operators to view the MAPE loop models in operation. In addition, operators can define properties that are checked at runtime. The ActivFORMS runtime environment supports four types of properties: (1) $A[]\ !Mx.Ly$, i.e. location Ly of model Mx is never reached; (2) $A[]\ bexpr1\ op\ bexpr2$, i.e. the operation on two boolean expressions $bexpr1$ and $bexpr2$ should hold always, with op being a logical operator (&, ||, etc.) or a relational operator (<, >, etc.); (3) $Mx.Ly{-}{>}Mu.Lv$, i.e. if location Ly of model Mx is reached then location Lv of model Mu should be reached without time delay (the runtime environment verifies this property by checking execution traces); and (4) $Mx.Ly{-}{>}(t)\ Mu.Lv$ which is similar to property (3) but with a time constraint defined by time t.

Example 12. We illustrate the inspection of the executing feedback loop model for DeltaIoT. Figure 6.15 shows the GUI of ActivFORMS with a model of DeltaIoT in execution and a set of properties that are verified at runtime.

The main user interface window shows properties of the second, the third, and the fourth type (the first type does not apply to the DeltaIoT MAPE models). The coloured circle on the left side to each property indicates the status of the property: green indicates that the property holds, red indicates a violation. The status of the properties over time can also be inspected via a log file. The model visualisation window shows the analyser model in execution. The snapshot is taken at the time when the verifier is performing a verification (state *RuntimeVerfication*).



(a) Main user interface



(b) Model visualization

Figure 6.15: GUI of ActivFORMS in action for DeltaIoT

6.4.4 Stage 4: Evolution of Feedback Loop and Adaptation Goals

The goal of the fourth stage is to support on the fly changes of the adaptation goals and MAPE models. ActivFORMS support these runtime changes through the goal management layer, see Fig. 6.12. We focus in this section on adding a new adaptation goal and update the feedback loop models accordingly. The approach for changing an existing goal or updating a feedback loop model for other purposes (e.g., correct a bug or change one of the MAPE models) is similar. The input elements of Stage 4 are: (1) a new adaptation requirement, (2) the MAPE-K feedback loop models that are currently in use; (3) the Uppaal model checker, and (4) the ActivFORMS runtime environment equipped with the user interface and online update plug-ins.

Recall that in ActivFORMS, we assume that the probes and effectors are available to monitor and adapt the managed system as needed for the adaptation goals. On the fly updates of adaptation goals and the feedback loop model that require an update of the managed system itself, including the probes an effectors it provides, is out of scope of this paper.

Stage 4 consists of three main engineering activities: *specify new adaptation goal and quality model*, *update and verify feedback loop model*, *enact feedback loop model*. The first and second activity are performed offline, the third activity is performed on the running system using the components of the goal managing layer. We explain now each of these activities.

6.4.4.1 Specify New Adaptation Goal and Quality Model

Support for changing adaptation goals during operation is considered a key aspect of self-adaptation [47, 133, 175, 203]. However, there is limited research in this area. With its model-driven and modular approach, ActivFORMS provides first-class support for on the fly changes of adaptation goals. When a new requirement appears, two steps are requited: (i) the new requirement needs to be translated to an adaptation goal, and (ii) a quality model needs to be specified together with

a verification query. Once deployed, the analyzer will use the quality model and query to perform an analyse of the adaptation options. The planner will use the new adaptation goal to select the best adaptation option. We discuss the updates of the MAPE models below.

Example 13. We illustrate how latency is taken into account in DeltaIoT as an additional quality property. Recall from Section 6.3 that DeltaIoT has a third additional requirement that needs to be activated at runtime: *R3. The average latency of messages should be less than 5% of the cycle time.* This new requirement is translated to an adaptation goal as follows:

Listing 6.12: Definition of adaptation goal for latency

```
type struct {
  ...
  int latency;
} Qualities
int MAX_LATENCY = 5;
bool satisfactionGoalLatency(Configuration gConf, int MAX_LATENCY) {
  return gConf.qualities.latency <= MAX_LATENCY;
}
```

Figure 6.16 shows the quality model to estimate latency for DeltaIoT.

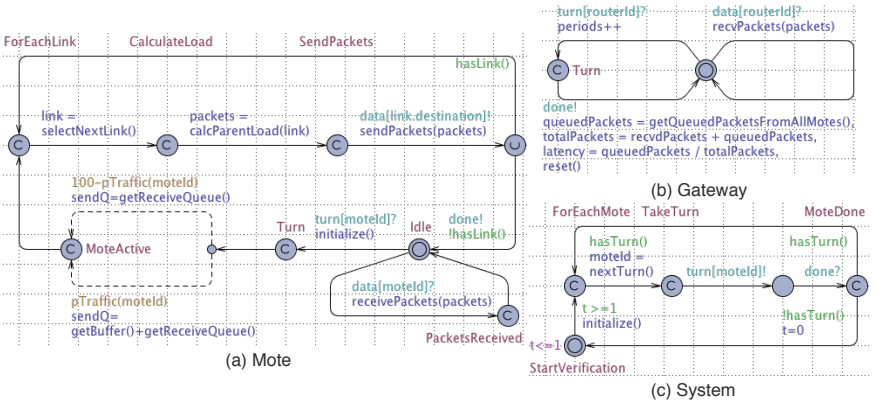


Figure 6.16: Latency model for DeltaIoT

The model has a similar structure as the quality model to estimate energy consumption (that is available at the project website). For the verification of the model, we use the following query:

Listing 6.13: Verification query for latency model

```
simulate 30[<=1](<>Gateway.latency)
```

The query calculates per adaptation option the estimated latency for 30 simulation runs (guaranteeing an RSEM of 5%). The adaptation options are the same as for the other quality models (see Example 10). The *System* automaton activates the motes one by one (*moteId = nextTurn()*). Each *Mote* can then send messages to its parents in the time slots dedicated to it (*sendPackets(packets)*). When the *Gateway* gets it turn, it computes the latency based the proportion of messages that did not

arrive (i.e., the messages that remained in queues) compared to the total number of messages (i.e. the messages in the queues plus the messages that arrived). For each simulation run the verifier assigns the uncertainty value for *pTraffic(moteId)* per mote. The analyzer model uses the 30 results to compute and estimated average latency with the required accuracy.

6.4.4.2 Update and Verify Feedback Loop Model

Adding a new adaptation goal requires updates of the MAPE models. Typically, the monitor model needs to be extended with support to track the new quality property and possibly new uncertainties that relate to the adaptation goal. The analyser needs to be extended with support to perform analysis of the new quality property. To that end, the analyser relies on the statistical model checker to verify the the new quality model using the verification query (both specified in the first activity). The planner needs to incorporate the new adaptation goal in the existing set of goals and use these updated set of goals to select the best adaptation option. In addition, new types of plan steps may need to be incorporated in the planner. Finally, the executor may need to be extended to be able to deal with new types of plan steps.

Once the new elements are incorporated in the MAPE models, they need to be verified to ensure their correctness. To that end, the initial stub models of probes, effectors, and the verifier need to be updated to ensure that they feed the MAPE models with proper input to check the correctness of the MAPE models. For the verification, the initial verification properties can be checked again. The generic properties that require an instantiation for the domain at hand ($Pr5$, $Pr6$, and $Pr11$) may need to be extended. Finally, if necessary, additional domain specific properties may need to be defined and verified. For updating and verifying the MAPE models, we use the Uppaal tool.

Example 14. We illustrate the updates of the DeltaIoT models that are required to deal with the latency goal. Figure 6.17 shows the updated analyser model. For the other updated MAPE models, we refer to the project website.

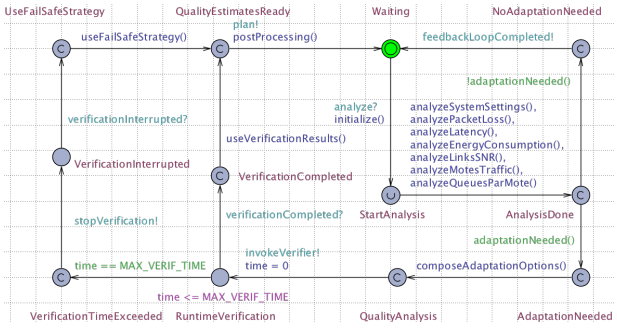


Figure 6.17: Updated analyzer model for DeltaIoT to deal with latency

The *Analyzer* model is extended with two functions: *analyzeLatency()* checks whether the latency of the network is above the threshold of maximum latency,

and *analyzeQueuesPerMote()* checks whether the queues for each mote are saturated or not. This data is taken into account when evaluating *adaptationNeeded()*. Furthermore, when the analyser invokes the verifier it will perform analysis of the latency model in addition to packet loss and energy consumption.

For the verification of the updated MAPE models, we could reuse the initial set of properties (see Example 5). As the changes of the models were limited, the time to perform verification with Uppaal increased on average with less then 2%. Detailed results are available at the project website.

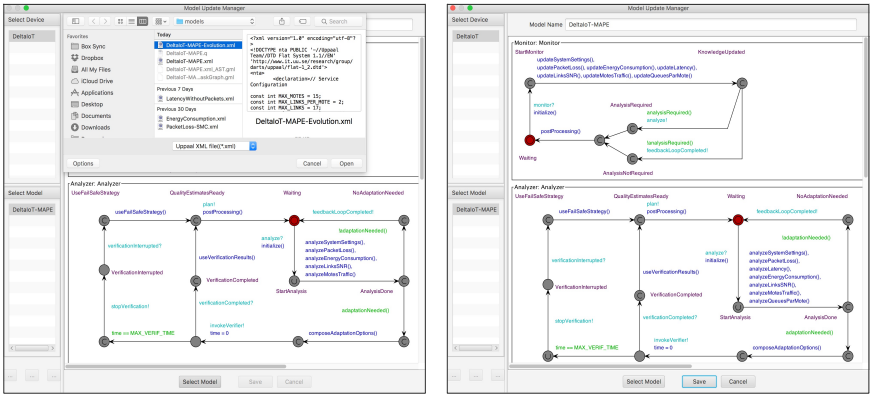
6.4.4.3 Enact Feedback Loop Model

The final activity to evolve the feedback loop is to update and enact the running models. Updating a running feedback loop model with ActivFORMS follows the classical process of runtime updates based on quiescence [123]. A quiescent state of a component is a state where no activity is going on in the component so that it can be safely updated. In ActivFORMS the starting states (*Waiting*) of each MAPE model is the default quiescent state because the MAPE behaviours wait in these states to be triggered to start their respective adaptation functions (otherwise, there is some adaptation activity going on).

As explained in Section 6.4.2.1, in ActivFORMS, goal management is supported by a user interface and an online update component. The procedure to update a running model is as follows:

1. The model update is loaded via the user interface;
2. The online update component tracks when the MAPE models enter quiescence states;
3. Once the models are in quiescence states the online update component notifies the virtual machine to start updating the running feedback loop model;
4. The virtual machine halts the execution of the running feedback loop model and all incoming signals are put into a waiting queue;
5. The virtual machine saves the state of the old model;
6. The virtual machine loads the updated MAPE models; new knowledge models are also loaded;
7. The state of the corresponding variables from the old models is copied to the new models. New variables are initialised;
8. The virtual machine starts executing the new model;
9. Pending signals waiting in the queue are processed (first-in-first-out).
10. Normal execution continues.

Example 15. We illustrate the update of the MAPE models of DeltaIoT to incorporate the latency goal. Figure 6.18 (left) shows the ActivFORMS user interface to perform on-the-fly updates of the feedback loop model and adaptation goals. The window shows how the *DeltaIoT-MAPE-Evolution* model is selected for update. The window on the right shows that updated models of the *Monitor* and *Analyser* that are ready to start execution taking into account the latency goal.



(a) Model update dialog (b) New model updated

Figure 6.18: Live updates of the MAPE models for DeltaIoT (user interface to perform on-the-fly updates left hand side; the updated models ready for execution after the update right hand side).

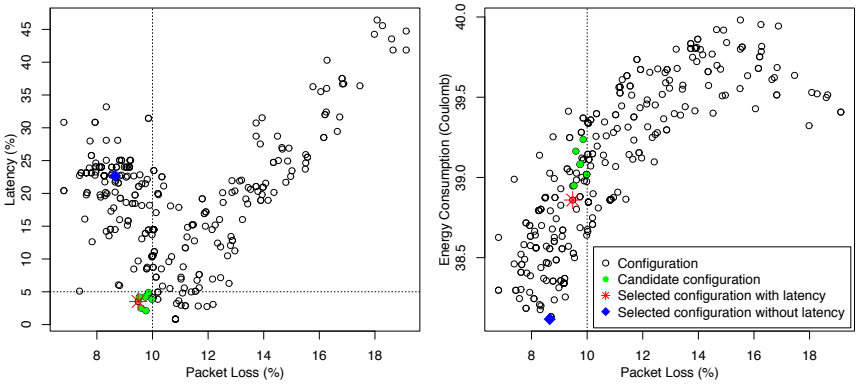


Figure 6.19: Selection of the best adaptation option with three adaptation goals.

Figure 6.19 shows the selection of the best adaptation option with three adaptation goals.

The diagram on the left hand side shows the adaptation options at a particular point in time. The options that comply with the adaptation goals for packet loss ($<= 10\%$) and latency ($<= 5\%$) are marked in green plus one in red. The diagram on the right hand side highlights the same adaptation options; the red option is selected for adaptation. The selected adaptation option has an expected packet loss of 9.5%, a latency of 3.5%, and an energy consumption of 38.9.

The adaptation option marked in blue in both figures shows the option that would have been selected if the latency goal would not have been taken into account. The figures illustrate that ensuring the latency goal may introduce a small tradeoff against the other two adaptation goals.

6.5 Evaluation of ActivFORMS

We evaluate ActivFORMS using the DeltaIoT network deployed at KU Leuven shown in Figure 6.2. The default setup consists of 15 motes, each mote comprising two elements: (1) a Raspberry Pi that is responsible for sensing, local processing, and network management operations, and (2) a RN2483 LoRa module[1] that is in charge of radio communication. The output power of the LoRa module is 20.2 mA for power setting 0 and 38.9 mA for power setting 15 at 3.3 V. The gateway runs on a regular server machine that is responsible for processing network data and storing the network statistics in a database. The server offers an API to a probe and effector to monitor and adapt the network. For the simulation tests we used a Macbook with 2.5 GHz Core i7 processor, and 16 GB 1600MHz DD3 RAM. All the data of the evaluation is available at the project website, including a link to the DeltaIoT artifact [111] that can be used to replicate the experiments.

Unless mentioned differently, we run the default setup of the DeltaIoT network with 15 motes for a period of 24 hours. The cycle time is set to 9.5 minutes, corresponding to 153 cycles in 24 hours. A cycle consists of two phases: the first 8 minutes are allocated to the motes to communicate date downstream to the gateway; the remaining 1.5 minutes are allocated for the communication of adaptation messages from the gateway upstream to the motes. The maximum verification time is set to 8 minutes. Each mote can generate 10 messages per cycle, subject to its traffic load profile, see the description in Section 6.3. In each cycle, each mote gets 40 slots of 2 seconds for communication. The size of the *send-queue* is 60, which implies that messages in the queue are sent within two cycles. Messages from children that arrive when the *receive-queue* is full are discarded. The values for SNR are based on the actual conditions of the wireless communication. The spreading factor of all motes is fixed (set to 8) in all experiments. In simulation mode, the cycle time is set to 12 minutes with communication slots of 4 seconds, and the spreading factor is set to 11. The values for uncertainties of SNR and traffic load are based on empirical data as described in Section 6.3.

The evaluation consists of four parts. In part one, we compare the tradeoff between the accuracy of the verification results with the time required for verification at runtime. In part two, we compare ActivFORMS with two other approaches: an over-provisioning approach that is common in practice and an approach that uses runtime quantitative verification (RQV [37]). In the third part, we test scalability. In particular, we measure adaptation time and memory usage for network configurations with an increasing number of motes. We compare the scalability of ActivFORMS with RQV. Finally, in the fourth part, we dynamically incorporate the latency goal in the DeltaIoT network with ActivFORMS and test the impact of it. The tests in parts two and four are performed on the physical network; the test in parts one and three are performed in simulation.

[1] http://ww1.microchip.com/downloads/en/DeviceDoc/50002346C.pdf

6.5.1 Tradeoff Between Accuracy and Adaptation Time

To evaluate the tradeoff between the accuracy of the verification results and the adaptation time (which is primarily determined by the time required for verification), we used a network setup with 15 motes and two adaptation goals: energy consumption and packet loss. We run two experiments: in the first we measure the tradeoff between accuracy of the verification results and the verification time; in the second, we evaluated the quality of adaptation decisions for different settings.

Results. In the first experiment, we picked a random adaptation option and applied verification for both qualities. The graphs in Figure 6.20 plot the results of 1000 runs. The results for energy consumption (graphs on the left hand side) show that lower values for RSEM (i.e., more simulation runs) result in more accurate verification results; e.g., the quartiles of the boxplots for $RSEM$=2% are -0.52/+0.45 *Coulomb* compared to -0.18/+0.11 *Coulomb* for *RSEM*=0.5%. The tradeoff is an increase in verification time; the median for *RSEM*=2% is 2 ms compared to 16 ms for *RSEM*=0.5%. The results for packet loss are similar; more accurate verification results (E defines the approximation interval $p \pm \epsilon$ and A defines the confidence $1 - \alpha$) require more verification time.

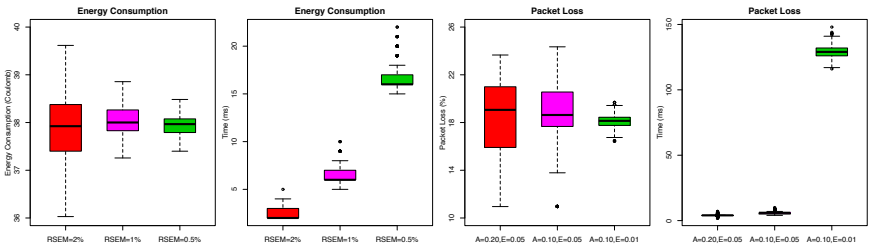


Figure 6.20: Tradeoff between accuracy and adaptation time with ActivFORMS

In the second experiment, we evaluated the quality of adaptation decisions for a simulation run of 24 hours for the different settings. Figure 6.21 shows the results. The boxplots show that settings that produce more accurate verification results also produce better adaptation decisions. For example, for a setting with A=0.20, E=0.05 and *RSEM*=2% the median for packet loss is 9% and for energy consumption it is 37.3, compared to 7% and 37.0 respectively for a setting with A=0.10, E=0.01 and *RSEM*=0.5%. The cost is an increase of adaptation time from median 9.2 s to 19.9 s.

Conclusions. Runtime statistical model checking allows the user to set the accuracy of the verification results (approximation interval and confidence). The test results show that more accurate results (smaller approximation interval and higher confidence) produce better adaptation decisions. However, the cost is an increase of adaptation time.

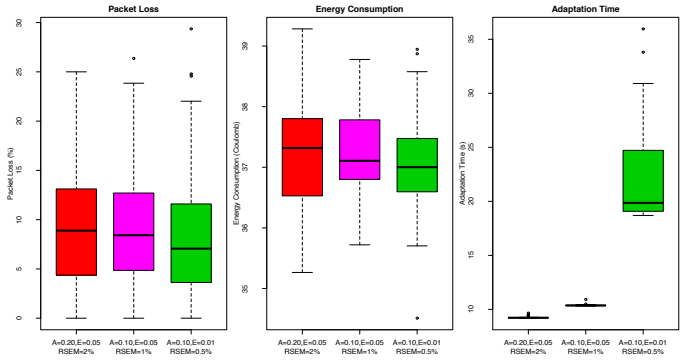


Figure 6.21: Impact on quality properties for different verification settings

6.5.2 Comparison ActivFORMS with Reference Approach and RQV

For the experiments in part two, we consider two adaptation goals: R1. Average packet loss should be below 10% and R2. Energy consumption should be minimised. For R1, we used an approximation interval with $\varepsilon = 0.01$ and confidence $\alpha = 0.10$, i.e., 90%. For R2, we used an RSEM of 0.5%, which requires 30 simulation runs (see explanation in Section 6.4.3.2). We compare ActivFORMS with a reference approach where all motes communicate at maximum power and send all message to all their parents. This over-provisioning approach is common in practice to assure high packet delivery performance at the cost of the lifetime of the network. We also compare ActivFORMS with a state of the art analysis approach that uses RQV. For RQV, we translated the automata models for the quality properties to a Discrete Time Markov Chain model for packet loss and a Markov Decision Process model for energy consumption. For the verification at runtime we enabled the analyser model to use the PRISM model checker [126] with the default settings. For the definition of the RQV models, we refer to the project website.

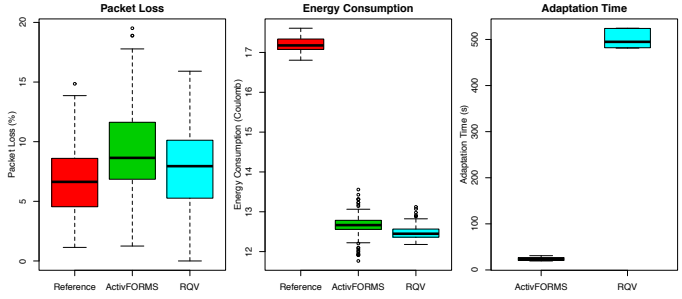


Figure 6.22: Results for a DeltaIoT setting with 15 motes and two adaptation goals

Results. Figure 6.22 shows the results for runs of 24 hours, where adaptation is applied every cycle of 9.5 minutes. The reference approach is able to realise

$R1$ with a better result for packet loss as ActivFORMS (median of 6.6% versus 8.6% for the ActivFORMS). However, ActivFOMRS significantly reduces energy consumption with about 27% (12.5 Coulomb compared to 17.2 for the reference approach). RQV on the other hand realises a better result as ActivFORMS on packet loss (median of 7.9%) and energy consumption as ActivFORMS (average 12.4 Coulomb). Although RQV applies exhaustive verification, due to time constraints (max 8 minutes verification time), the approach was able to verify only a fraction of the possible adaptation options and hence is not able to find the best solution. The right hand side of Figure 6.22 shows the adaptation times that are almost completely used for verification. On average, ActivFORMS used 23.7 seconds to compute the verification results and realise adaptation. RQV on the other hand, used the complete available time slot for verification, resulting in an average adaptation time of 495 seconds, i.e. 8.3 minutes.[2]

Statistical analysis of the data shows that there is no significant difference for average packet loss between ActivFORMS and RQV ($p = 0.9993$ using the paired t-test[3]). The average energy consumption with ActivFORMS is slightly higher compared to RQV ($p < 0.00005$ using the Wilcoxon signed-rank test). However, relatively this increase is very small (the mean value of average energy consumption with ActivFORMS is 12.7 compared to 12.4 for RQV).

Conclusions. Self-adaptation with ActivFORMS guarantees, with sufficient accuracy and confidence, the packet loss requirement and improves on energy consumption with about 27% compared to the reference approach. The approach is able to realise self-adaptation within a time that is only a small fraction of the cycle time of 9.5 minutes. On the other hand, RQV realises slightly better results for both requirements, but it requires significantly more time to do so.

6.5.3 Scalability of ActivFORMS Compared with RQV

To evaluate the scalability of ActivFORMS and compare it with RQV, we measured the adaptation time and the memory usage for network settings with increasing complexity. Concretely we increased the number of motes of the IoT network from 5 to 25 in steps of 5. In each step, the number of adaptation options increased based on $6^{\frac{m}{5}}$ with m the number of motes; e.g., a setting with 10 motes has 36 options, while a setting with 25 mote has has 7776 options. We applied adaptation for packet loss (approximation interval with $\varepsilon = 0.01$ and confidence $\alpha = 0.10$) and energy consumption (RSEM of 0.5%). All the models used for the experiments are available at the project website.

Results. Figure 6.23 shows the results for ActivFORMS on the left hand side and RQV on the right hand side. The graphs show the results of 100 runs, each based on a random selected adaptation option for each of the IoT network from 5 to 25 motes. The results show that ActivFORMS scales well for networks up to 20 motes, both for verification time and memory usage. If we extrapolate the verification time for a setting with 20 motes (median $168\,ms$) that has 1296 adaptation

[2] Analysis was interrupted when the verification of the last option that was started within 8 min completed.

[3] We used the Anderson-Darling test to select appropriate statistical tests.

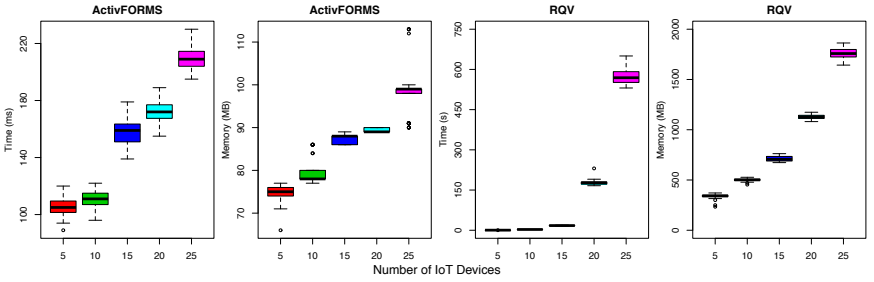


Figure 6.23: Results of the scalability tests

options, the total verification time would be $3.63\,min$, which is less than half of the available 8 minutes. For the configuration with 25 motes with 7776 adaptation options, the total verification time would be $26.44\,min$. However, if we relax the verification settings slightly (approximation interval with $\varepsilon = 0.5$ and confidence α =0.10, and RSEM of 1%), the verification time with ActivFORMS decreases to $6.48\,min$ and the solution scales well, although with a bit less accurate results. ActivFORMS requires between 75 and 100 MB memory for verification. RQV on the other hand does not scale for more complex networks. An extrapolation of the verification time for 15 motes (median $17.45\,s$) with 216 adaptation options, would require around $62.82\,min$, which is 8 times higher as the available time of $8\,min$. In addition, RQV requires 500 to 1800 MB memory for verification.

Conclusions. The test results show that ActivFORMS scales well for IoT network settings with 25 motes and up to 10 K adaptation options. With RQV complete verification is limited to settings with 10 motes. Furthermore, RQV requires up to 18 times more memory as ActivFORMS.

6.5.4 Dynamically Incorporating Latency Goal

In part four, we dynamically add a latency goal to the running system and evaluate the impact of it on the quality properties and the adaptation time. We used the same setup as in part two with 15 motes and approximation interval with $\varepsilon = 0.01$ and confidence $\alpha = 0.10$. The test started with the packet loss and energy consumption goals only. After 24 hours, the latency goal was dynamically added for another 24 hours. The latency model and the approach to add the goal dynamically is explained in Examples 14 and 15.

Results. Figure 6.23 shows the test results. As we can see, adding the latency goal drastically reduces the latency (median $0.00\,\%$ of the cycle time with quartiles ± 0.00 compared to $30.80\,\%$ and -12.1/+5.8 for the ActivFORMS setup without latency goal. This improvement has only a small effect on packet loss (median decreased from $6.64\,\%$ to $7.95\,\%$) and energy consumption (median increased from 12.66 to 12.80 *Coulomb*). The verification of the latency model increased the adaptation time from with 23.72 sec to 45.78 sec.

We also performed statistical analysis on the data of the reference approach and ActivFORMS. For average energy consumption, ActivFORMS is significantly better compared to the reference approach ($p < 0.00005$ using the Wilcoxon signed-

rank test). The average packet loss of ActivFORMS is slightly higher compared to the reference approach ($p < 0.00002$ using the paired t-test). However, relatively this increase is very small (the mean value of average packet loss with ActivFORMS is 0.07 compared to 0.08 for the reference approach). For the average latency on the other hand, ActivFORMS clearly outperforms the reference approach ($p < 0.00005$ using the paired t-test).

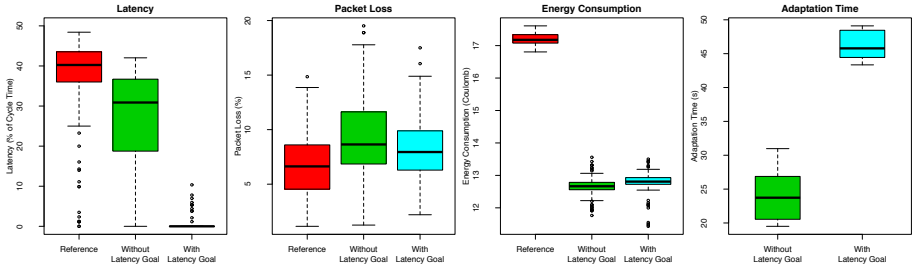


Figure 6.24: Impact of dynamically adding a latency goal

Conclusions. ActivFORMS provides first-class support for dynamic updates of the adaptation logic. The test results show that adding a latency goal drastically reduces latency of packet delivery. There is a small decrease in packet loss and a small increase in energy consumption. However, about three times more adaption time is required for the verification of the newly added goal.

6.6 Related Work

A vast body of work exists on assurance techniques for self-adaptive systems, for recent overviews see [49, 58, 135, 213]. Aligned with the research presented in this paper, we focus on approaches that use formal techniques to provide assurances. We have structured related work in three groups: approaches that provide assurances at design time, runtime approaches, and hybrid approaches. For each group, we discuss a selection of representative approaches, based on classic work and more recent work. The section concludes with a summary that positions ActivFORMS in the current landscape of research.

Design time approaches. [112] use Petri Nets to model adaptive systems; the models are automatically translated to executable programs. Properties specified in linear temporal logic (LTL) allow verifying invariants and constraints about the system and its goals, e.g., "if adaptation is triggered, eventually it will be applied," or "the adaptive program should tolerate 2-packet loss throughout its execution." Conformance between the models and programs is guaranteed using model-based testing. [9] deals with partial knowledge by automatically producing service-oriented systems in two phases. The first phase (elicit) applies a technique called StrawBerry that takes service descriptions to derive behaviour automata of the service interactions. The second phase (integrate) takes the automata to automatically synthesise a service choreography that satisfies the system goal. The

approach relies on tools to guarantee functional correctness-by-construction. [147] presents a general approach to specify correctness criteria for the dynamic update of a system and a technique to automatically compute a controller that handles the transition from the old to a new specification. The approach syntheses a controller that guarantees progress towards the update and performs a safe update, i.e., by guiding the system to a safe state in which the update can start, ensuring that the update will eventually occur and satisfy the new specification. [43] proposes an approach for evaluating the resilience of self-adaptive systems by applying robustness testing techniques to the controller to uncover failures that can affect system resilience. The approach, that is based on probabilistic model checking, quantifies the probability of satisfaction of system properties when the target system is subject to controller failures. The responses to malformed input between controller and target system are used to classify robustness.

The related approaches in the first group provide guarantees based on the principle of correctness-by-construction. Consequently, the guarantees are based on the knowledge available at design time. With ActivFORMS, correctness-by-construction is applied at design time to provide guarantees for the functional correctness of the feedback loop. These guarantees are complemented with guarantees for quality goals obtained during operation based on data of uncertainties collected at runtime.

Runtime approaches. [37] uses a runtime probabilistic model of an adaptive system and applies runtime quantitative verification (RQV) to identify and enforce optimal system configurations under changing conditions. Performance and reliability goals are expressed as probabilistic temporal logic formulae. The MAPE components exploit different tools to assure the quality goals. Techniques, such as cashing and lookahead, can be used to improve the efficiency of RQV. [69] presents a quantitative approach for making adaptation decisions under uncertainty, called POISED. POISED builds on possibility theory (that is grounded in fuzzy mathematics) to assess both the positive and negative consequences of uncertainty. At runtime, POISED makes adaptation decisions, i.e., runtime reconfigurations of its customisable software components, that result in the best range of potential behaviour, improving the system's quality of service. [75] proposes an approach that relies on the mathematical foundation of control theory. The approach automatically learns a system model and synthesises a controller at runtime, providing control-theoretic guarantees for stability, overshoot, setting time and robustness of system operating under disturbances, and this for one goal (setpoint). A Kalman filter and a change point detection mechanism are used to update the system model on the fly. [145] applies proactive adaptation under uncertainty. The approach uses a probabilistic model of the adaptive system in which the adaptation decision is left underspecified through nondeterminism. At runtime, a probabilistic model checker resolves the nondeterministic choices so that the accumulated utility over a horizon is maximised. The adaptation decision is optimal over the horizon and takes into account the inherent uncertainty of the environment predictions needed for looking ahead.

The related approaches in the second group focus on guarantees for quality goals obtained at runtime. These guarantees are derived from additional knowledge obtained during execution. While existing work primarily relies on exhaustive verification, which is time and resources demanding, ActivFORMS relies on statistical model checking to provides guarantees for quality goals at runtime, which is more efficient, though at the cost of some reduction of accuracy. In addition, ActivFORMS also provides guarantees for the functional correctness of the feedback loop.

Hybrid approaches. FLAGS [13] proposes a goal-driven approach for self-adaptation that spans design and runtime. The approach supports modelling both crisp goals specified in linear temporal logic and fuzzy goals specified in fuzzy temporal language. These models can then used at runtime to monitor goal violations that trigger a modification of the goal model to enforce adaptation on the running system. Related approaches are RELAX [217] that offers a textual language for specifying requirements with first-class support for uncertainty, and [175] that distinguishes between "awareness requirements" that describe the situations that require adaptation and "evolution requirements" that prescribe what to do in these situations. [200] offers a domain-specific language to model feedback loops and their interactions. At design time feedback loop models are specified by means of operations, runtime models, and interactions. An additional layer diagram specifies the interactions between the feedback loops and the managed system. At runtime, EUREMA offers an interpreter that directly interprets the models to realise adaptation. Additionally, the models can be dynamically adjusted, supporting evolution. [73] offers a mathematical framework for efficient run-time decision making in two steps. At design time a pre-computation is applied taking a model of the system and desired goals to generate a partially evaluated set of symbolic expressions that represent verification conditions to be satisfied to meet the requirements. At runtime, the actual values are bound to the variables enabling the expressions to be evaluated efficiently. The focus of the work is on quality requirements, such as reliability or energy consumption. [30] applies obstacle analysis, i.e., a goal-oriented form of risk analysis whereby obstacles to system goals are identified, assessed, and resolved through countermeasures. During requirements engineering, obstacle/goal trees are specified together with predicates that determine the satisfaction rates of probabilistic goals. At runtime, the system is monitored and the satisfaction rate of high-level goals is determined. When goals are not satisfied, alternative countermeasures are selected and the goal model is updated. The running system is then adapted according to the selected countermeasures.

The related approaches in the third group combine assurance techniques at design time and runtime to provide guarantees for the system goals. In addition to specific differences, such as efficient decision making at runtime, ActivFORMS integrates design time and runtime guarantees with first-class support for dealing with changing adaption goals and updating feedback loop models. The most closely related approach is EUREMA [200]. However, this appraoch has no formal basis and consequently cannot provide the guarantees that ActivFORMS can give.

Position of ActivFORMS in state of the art landscape. Table 6.1 summarises the selection of representative related work and compares it with ActivFORMS. The use of formal techniques in self-adaptive systems has gained increasing attention in recent years [135, 185, 213]. We observe that quantitative approaches are currently dominant. In contrast to existing work, ActivFORMS is able to provide guarantees for both the functional correctness of the feedback loop and the quality goals of the adaptive system. By using a statistical approach, ActivFORMS is able to provide guarantees for quality goals in an efficient way. There is a tradeoff in accuracy, but the approach allows to set this tradeoff as required. In addition, ActivFORMS provides first-class support for changing the adaptation goals and the feedback loop on the fly.

6.7 Conclusions and Future Work

Guaranteeing compliance of the adaptation goals in self-adaptive systems is challenging since the uncertainties can appear at any time. To tackle this challenge, we presented ActivFORMS (Active FORmal Models for Self-adaptation), a formally founded model-driven approach for engineering self-adaptive systems. ActivFORMS contributes to the state of the art an approach that provides: 1) functional correctness of the feedback loop by direct execution of formally verified models of the feedback loop using a reusable virtual machine, 2) efficient guarantees for the adaption goals with a required level of confidence using statistical model checking at runtime, and 3) support for changing adaptation goals on the fly and updating of verified feedback loop models that meet the new goals. We evaluated ActivFORMS with a real world IoT application deployed at KU Leuven. The test results demonstrate that ActivFORMS can provide the required guarantees for the stakeholder goals for a realistic IoT setup with 15 motes that is subject to different types of uncertainties. Scalability tests show that ActivFORMS scales well to setups with up to 25 motes, which cannot be handled by a state of the art exhaustive runtime quantitative verification approach.

In future work, we plan to study how ActivFORMS can be applied to adaptation problems with more complex types of uncertainties, such as uncertainties related to the structure of models. We also plan to study how ActivFORMS can be applied in systems that require multiple feedback loops that need to work together to solve an adaptation problem.

Table 6.1: Summary of selection of related work and comparison with ActivFORMS

Class	Summary approach	Functional correctness	Guarantees for QoS	Runtime changes of goals
Design time	Petri Net models of adaptive system that are automatically translated to a program [112]	Verification of invariants and constraints of adaptive program specified in LTL		
	Automata of services are automatically translated to a service choreography [9]	Service choreography functionally correct-by-construction		
	Automatic synthesis of a controller that guides a system to a safe state and performs an update [147]	Correctness criteria to progress towards a safe state and perform the update		Synthesised controller performs updates of the system with potentially new goals
	Robustness testing with malformed input to uncover controller failures that can affect system resilience [43]	Probabilistic model checking of system properties when system is subject to controller failures		
Runtime	Probabilistic runtime model of adaptive system analysed using RQV [37]		Runtime verification of quality goals expressed in probabilistic temporal logic	
	Builds on possibility theory to make adaptation decisions under uncertainty [69]		Assesses the consequences of uncertainty to reconfigure system improving its OoS	
	Automatically learns/maintains a system model and synthesises a controller at runtime [75]		Control-theoretic guarantees for setpoint goal of system that operates under disturbances	
	Proactive adaptation using a probabilistic model that captures adaptation decisions through nondeterminism [145]		Model checker resolves adaptation decisions maximising accumulated utility over a horizon	
Hybrid	Goal model with crisp and fuzzy goals kept alive at runtime to enforce adaptations [13]	Reasoning over goals specified in LTL and fuzzy temporal logic	Goal violations monitored at runtime trigger countermeasures to adapt system	
	Domain-specific language to model feedback loops that are directly interpreted [200]	Syntactic correctness of models based on supporting tools		Feedback loop models can be dynamically updated
	Partially evaluated set symbolic expressions are completed at runtime to satisfy the requirements [73]		Variables of symbolic expressions are bound to values to realise quality goals at runtime	
	Obstacle analysis to identify, assess, and resolve goal obstacles through countermeasures [30]	Predicates allow determining the satisfaction rates of probabilistic goals	Not satisfied goals change the goal model, which in turn adapts the running system	
ActivFORMS	Verified feedback loop model that is directly executed exploits statistical model checking to make efficient adaptation decisions	Design time verification of TCTL properties to guarantee the correctness of feedback loop model	Runtime statistical model checking to guarantee the adaptation goals with required confidence	Goal management enables on the fly updates of adaptation goals and feedback loop model

Chapter 7

Conclusion and Future Work

In this last chapter, we summarise the contributions of our research, we report lessons learned from our experience, and outline a number of interesting paths for the future research.

7.1 Conclusion

Many modern software systems have to deal with uncertainties at runtime. The uncertainties originate from inadequate knowledge about the system at design time, changing conditions in the operation conditions of the system, variations in user needs, etc. Without any control mechanism, such a software system may behave in undesired ways. Self-adaptation enables a software system to deal autonomously with such uncertainties. Despite a vast body of research in the area of self-adaptation, providing guarantees that adaptation goals are achieved is still a open challenge. This thesis contributes ActivFORMS, a formally founded model-driven approach for engineering self-adaptive software systems. The contributions of ActivFORMS to the state of the art are the following:

1. An innovative approach that guarantees functional correctness of the feedback loop using a reusable virtual machine that directly executes the formal models of the feedback loop. To support software engineers, we provided a set of templates that assist designers in designing feedback loops.
2. An efficient approach to provide guarantees for the adaptation goals of the system that combines simulation and statistical model checking at runtime. The approach enables to tradeoff the resources required to provide guarantees with the level of accuracy that is provided for the guarantees.
3. A novel approach to support changing adaptation goals on the fly by updating the verified loop models that meet the new goals. This approach relies on the principles of quiescence for the executing models of the feedback loop.

We applied ActivFORMS to multiple applications from different domains, including a real-world deployment of an IoT building security monitoring that was developed and evaluated in collaboration with VersaSense, an IoT company located in Belgium. The evaluations results show that ActivFORMS successfully provides the guarantees required by the stakeholders.

7.2 Lessons learned

We report some important lesson learned during this research.

7.2.1 Use of Formal Techniques

ActivFORMS relies on various formal techniques. Formal techniques are being used in software engineering for decades and we observe a similar trend in the area of self-adaptation [132, 213, 214]. One concern often associated to the use of formal techniques is the need for additional knowledge that is required to use them. Using ActivFORMS requires the formal specification of stochastic timed automata and properties expressed in computation tree logic as well as queries required for statistical model checking. Our experience show that these languages allow to specify a wide variety of solutions in a relatively intuitive way. Besides our research, we used ActivFORMS also in several Master courses. Feedback from the students confirm the usability of the modelling formalisms. However, further research is required to confirm this observation.

7.2.2 Expressiveness of Formal Languages

During our research, we gradually learned the power of the formal languages we used, but also some limitations in expressiveness. In our research, we relied on the UPPAAL tool suite that uses networks of timed automata as modelling formalism. Uppaal provides graphical modelling features combined with C-like language constructs. We used these languages constructs for modelling feedback loops of many applications to evaluate research results. We also used them for assignments in Master level courses. Although we have not found any particular restrictions in modelling feedback loops, we experienced limitations of the modelling language. Examples are limited support for rich types, restrictions in reusability (compared for example to Object-Oriented concepts), lack of libraries, and restrictions associated with the communication through signals. Note that these limitations are related to the specific formal languages we used in our research.

7.2.3 Scalability

Formal methods are often criticised for their limited scalability. This limitation relates to the required resources such as processing power, memory and time. The state space grows exponentially with the states in the system, causing the well-known state explosion problem [53]. In ActivFORMS, we minimized this effect using two strategies. At design time, we applied a modular approach where we used appropriate stub models of the managed system and the environment, allowing efficient verification of the correctness of the MAPE components. At runtime, we combined simulation with statistical model checking that are both known to be less resources demanding, although at the cost of bounds on the accuracy of the results.

7.3 Limitations

We report limitations of ActivFORMS that pave paths towards future work.

7.3.1 Limitations of the Simulation-based Approaches

ActivFORMS inherits the limitations of the simulation-based approaches, since runtime simulation and statistical model checking approaches are used to efficiently guarantee adaptation goals. There are a number of inherent limitations of the simulation-based approaches compared to the exhaustive verification [130]. One of the key limitations is related to the accuracy of the results. The simulation-based approach can not efficiently provide highly accurate result since the sample size will grows very large, which in turn have the reverse effect on the efficiency of the approach. Simulation based approaches also require special solutions in case of so called rare events.

7.3.2 Limitations of the Virtual Machine

The virtual machine is implemented in Java and is based on standard compiling techniques to execute the feedback loop models. The virtual machine has the following limitations:

1. The system is not suitable for adaptation of time critical systems. ActivFORMS allow to set logical time of a model to a physical time, which means that a tick of the model time can be set equivalent to milliseconds, seconds, minutes, or even hours. Internally, the virtual machine uses a Java timer that is invoked for each time tick. If the unit of the time tick is very small we might end up in a situation where the virtual machine could not manage to finish all the required computations in a particular time tick.
2. In our current implementation, the virtual machine is realized using Java technology that limit our approach to be used to settings where a Java virtual machine is available to run the ActivFORMS virtual machine.
3. As we discuss in the previous section, there are limits in the expressiveness of the formal language that is being used in our research. Earlier in this thesis (e.g., chapter 1), we also highlighted a number of restrictions regarding the types of updates of feedback loop models that ActivFORMS supports.

7.4 Future Work

We conclude with presenting interesting paths for future work starting from our research results. We start with challenges in the short term and then discuss long term challenges.

7.4.1 Short Term Challenges

Short term challenges closely connect with presented work.

7.4.1.1 Online Verification of Correctness Feedback Loop

So far, we achieved functional correctness of the feedback loop using offline verification of TCTL properties. Once the offline verification is completed, the virtual machine directly executes the formal model of the feedback loop according to the formal semantics of timed automata. However, our approach may have limitations in case a full verification of the feedback loop is not feasible before deployment. Examples are incomplete knowledge about the environment, constraints about available resources such as processing power and memory, etc. On the other hand at runtime, the virtual machine has full access to the model of the feedback loop. The virtual machine can exploit the available runtime information of the system to detect behaviour violations. The virtual machine can verify the formal model of the feedback loop along the path the system follows. ActivFORMS already support runtime verification of a subset of the timed computation tree logic (TCTL) properties, like constraints, safety, and liveness properties. We demonstrate how to use these properties in the section 6.4.3. Advanced support for online verification to automatically recover the system in case of an anomaly is an interesting path for future research.

7.4.1.2 Dynamic Calculation of RSEM

In chapter 5 and 6 we used the relative standard error of the mean (RSEM) to determine the number of simulations required for a quality model to estimate quality goals. Hence, the number of simulation required for a particular RSEM is based on offline experiments. Concretely, in the offline experiment, we take an average of the number of simulations required for different adaptation options to reach the required RSEM. But in reality, the RSEM may change dynamically due to variations in the uncertainty values and available adaptation options. Developing an automatic approach that dynamically determines and adapts the RSEM for simulation queries with a particumal accuracy under changing conditions is an interesting path for further research.

7.4.2 Long Term Challenges

We discuss two long term challenges.

7.4.2.1 Domain Specific Modelling Languages

One interesting topic for further research is investigating *domain-specific modelling languages (DSML)* for self-adaptation. A DSML can provide a high level of abstraction to specify feedback loops and adaptation goals, hiding low-level details of formalism. Furthermore, a DSML can be improve the efficiency for engineers to create self-adaptation solutions by using domain-specific abstractions. At runtime, a DSML model can automatically be converted into low-level formalism, such as timed automata, and directly executed using domain-specific interpreters. These domain-specific interpreters can provide domain-specific features that are not easy to be implemented otherwise. An example of the domain-specific interpreter is the virtual machine used in ActivFORMS approach that supports execution and update

of the formal model of the feedback loop(s). Some initial examples of approaches that use DSML in self-adaptation are presented in [91, 200].

7.4.2.2 Multiple Feedback Loops

Another line of research that to explore in the long term is to support coordination between multiple feedback loops that work together to solve an adaptation problem. Multiple feedback loops can be deployed locally handling different concerns, or distributed over different nodes/components of the software system working together. In this thesis, adaptation of software systems is restricted to only one feedback loop that interacts with the managed system and its environment. Supporting coordination between multiple feedback loops would open the possibility to realise self-adaptation in a decentralized setting. An interesting inspiring approach recently presented is [6].

Appendix A

A Model Interpreter for Timed Automata

Abstract

In the model-centric approach to model-driven development, the models used are sufficiently detailed to be executed. Being able to execute the model directly, without any intermediate model-to-code translation, has a number of advantages. The model is always up-to-date and runtime updates of the model are possible. This paper presents a model interpreter for timed automata, a formalism often used for modeling and verification of real-time systems. The model interpreter supports real-time system features like simultaneous execution, system wide signals, a ticking clock, and time constraints. Many existing formal representations can be verified, and many existing DSMLs can be executed. It is the combination of being both verifiable and executable that makes our approach rather unique.

A.1 Introduction

Model-driven development (MDD) is a software development methodology focusing on creating and exploiting domain models [167]. A domain model is an abstraction that describes selected aspects of a specific domain. An important part of MDD is the use of domain-specific modeling languages (DSML) [77]. Developers use DSMLs to efficiently build application models using elements of the domain and often express design intent declaratively rather than imperatively.

In a model-centric approach, models of the system are established in sufficient detail that the model can be executed, or used to generate executable code [167]. To achieve this, the models defined in a DSML might include, for example, representations of persistent and non-persistent data, business logic, and presentation elements. Integration to legacy data and services might require that the interfaces to those models are also modeled.

There are two common approaches to model execution. In the code-generation approach a DSML specified model can be translated to a program in a language like Java that can later be executed using the standard Java virtual machine. This approach works fine when the DSML is (roughly) a more abstract, richer version of an ordinary programming language. However, code-generation runs into trouble when the model has more declarative features like *simultaneous execution*, *system wide signals*, and *time constraints*, that is, model features that have no simple

counterpart in the target language into which it should be translated. Furthermore, model updates at runtime are basically impossible and any manual change in the generated code will ruin the connection to the model.

An alternative to code-generation is model interpretation that relies on the existence of a virtual machine able to directly read and run the model. The major advantage of this approach is that model updates at runtime are possible (see Section 5) and, as we will see in Section 3 and 4, the domain specific interpreter can provide support for model specific declarative features like the ones presented above. Having the model available at runtime also simplifies runtime verification of model dependent system goals (see Section 5).

The goal of this paper is to present a model interpreter for timed automata [4], first presented in [109]. Timed automata are an often used formalism to model real-time systems and it supports features like simultaneous execution, system wide signals, and time constraints[1]. Timed automata has a graphical representation suitable for humans and a corresponding XML based DSML suitable for machine processing. Formal properties (system goals) of models described by timed automata can be verified by a tool called UPPAAL [16]. In addition to handling real-time features, it is the use of a domain specific model being verifiable, executable in a real world scenario, and allowing model updates at runtime that makes our approach rather unique. See related work in Section A.6 for more details.

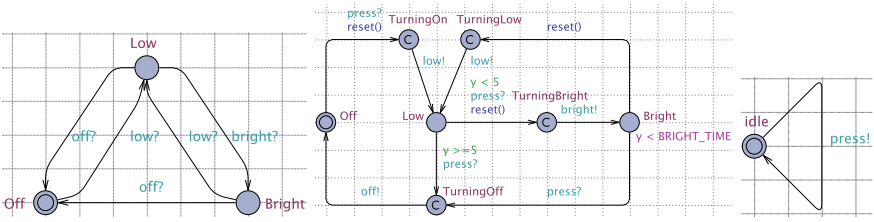
Timed automata will be presented in Section 2. The model interpretation is done in two steps: 1) The DSML defining the model is translated into an internal task graph based executable model (Section 3), and 2) A virtual machine, specifically designed for timed automata, interprets the executable model (Section 4). Step 1) is not novel since standard techniques from compiler design are used. The virtual machine on the other hand has novel features extending the functionality of a standard stack machine to handle a wide set of timed automata specific features. Additional features of our approach (e.g. support for runtime model updates and runtime verification) are discussed in Section 5. In Section 6 we present related work, and in Section 7 we present summary and conclusions.

A.2 Timed Automata

A timed automaton [4] is a finite automaton extended with a finite set of real-valued clocks. During a run of a timed automaton, all clock values increase with the same speed. The clock values can be compared to integers and these comparisons form guards that may enable or disable transitions and therefore constrain the automaton's behavior.

UPPAAL [19] is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata. UPPAAL comes with an XML based description language in which systems of timed automata can be defined, which is our DSML. UPPAAL also includes a number

[1] See the `uppaal.org` website for a list of industrial projects using timed automata and the UPPAAL verification tool.



(a) Lamp (b) Lamp Controller (c) User

Figure A.1: The simple lamp example.

of tools for visualizing the automata, simulation, and model verification. The aim of this section is to provide a brief introduction to timed automata as defined by the UPPAAL DSML. It can be considered as a brief (and informal) summary of the official UPPAAL tutorial [16], inspired by [97], with a focus on modeling and interpretation of timed automata. To simplify the presentation we use standard automata terminology (e.g. state, transition) rather than the standard timed automata terminology (e.g. location, fire an edge).

A.2.1 Networks of Timed Automata

A timed automaton is a finite-state machine extended with clock variables. All clocks progress synchronously. In UPPAAL, a system is modelled as a network of several such timed automata in parallel. The model is further extended with ordinary variables and the state of the system is defined by the state of all automata, the clock values, and the values of the variables. An automata may make a state transition separately or due to synchronization with another automata through channels. For example, for a channel x, a sender $x!$ can synchronize with a receiver $x?$ through a signal.

Figure A.1 shows three automata modelling a simple system with a lamp, a lamp controller, and a button to be pressed by a user. At start, when both the lamp and the controller are in state *Off*, if the user presses a button a signal *press!* is sent and the controller moves to state *TurningOn* due to synchronization *press?* followed by *LowLight* (sending a signal *low!*), and the lamp is turned on (due to *low?*). If the user presses the button again, the lamp is turned off. However, if the user is fast and within 5 time units presses the button twice, the lamp is turned on and becomes bright. The clock y of the lamp controller is used to detect if the user was fast $(y < 5)$ or slow $(y >= 5)$. The lamp stays bright for a certain period of time *BRIGHT_TIME* and then returns to Low state again.

We divide the models into two categories: *environment models* and *system models*. Environment models are used for simulation and enables offline verification of the system by providing input and getting output. For example, the user and lamp models are environment models in our lamp example. The system models are the models that contain the actual domain functionality/logic. In our lamp example, the lamp controller model is the system model.

The edges of the automata are annotated with three types of labels: a *guard*, expressing a condition (e.g. $y < 5$) on the values of clocks and variables that must be satisfied for the edge to be taken; a *synchronization* action (e.g. *press!*) which, when the edge is taken, forces a synchronization with other components on a complementary action, and an *update* (e.g. the function call $reset()$ which resets clock y to 0) defining actions to be taken when a transition is made. All three types of labels are optional: absence of a guard is interpreted as the condition $true$, and absence of a synchronization action indicates an internal (non-synchronizing) edge (e.g. *BrightLight* $\rightarrow$ *TurningLow* in the controller).

Only one state per automaton, called *control* or *active* state, is active at a time. States can also be annotated with *invariants* expressing constraints on the clock values for control to remain in a particular state. For example, the system can only remain in *BrightLight* as long as the value of y is less than *BRIGHT_TIME*.

UPPAAL defines two types of transitions between states: *action transition* and *delayed transition*. Action transitions can be further divided into *synchronization transition* and *internal transition*. If two complementary labeled edges (e.g. *press!* and *press?*) in two different automata are enabled then they can synchronize and a simultaneous *synchronization transition* is activated. In a delayed transition only the clock ticks and no actual state transition is made (e.g. *Bright* remains active in the controller while $y <$ *BRIGHT_TIME* and as long as no-one is pushing the button). Further progress in time might lead to an invariant violation ($y \geq$ *BRIGHT_TIME*) and an *internal transition* (*Bright* $\rightarrow$ *TurningLow*).

Finally, to enable modeling of atomicity of transition sequences in a given automaton (i.e. multiple transitions with no time delay) states may be marked as *committed* (indicated by a *c* in the circle). Commited states (e.g. *TurningOn* in the controller) make it possible to receive a signal (*press?* in *Off* $\rightarrow$ *TurningOn*) and send a signal (*low!* in *TurningOn* $\rightarrow$ *LowLight*) without any time delay.

A.2.2 The Timed Automata Modeling Language

The timed automata DSML is a straight forward XML markup of the transition graphs described previously. States (locations in UPPAAL) are nodes with a number of attributes (`id`, `name`, `commited`, `invariant`, etc.) and transitions are edges connecting source and target states (identified by their `id`s) with attributes (`guard`, `synchronization`, `assignment`) describing the transition conditions.

Figure A.2 shows an excerpt of the DSML for our lamp example. It starts with a section of global declarations `<declaration>` with variables and signals that are accessible anywhere in the system. In our lamp example, the global declaration section consists only of signal declarations, i.e., *press*, *off*, *low*, and *bright*.

The declaration section is followed by one or more template definitions describing a single automaton. Templates have a name (element `<name>`), a set of local variables and clocks (`<declaration>`), a set of states (`<location>`), an initial state (element `<init>`), and a set of transitions (`<transition>`).

The final DSML section, the *system declaration*, lists the *automata instances* planned to be used in the system. The system section is a description of how the

```
<?xml version="1.0" encoding="utf-8"?>
<nta>
    <declaration>chan press, off, low, bright;</declaration>
    <template>
        <name>LampController</name>
        <declaration>const int BRIGHT_TIME = 10;
            clock y;
            void reset(){ y = 0;}
        </declaration>
        <location id="id0">
            <name>TurningLow</name>
            <committed/>
        </location>
        <location id="id1">
            <name>Bright</name>
            <label kind="invariant">y &lt; BRIGHT_TIME</label>
        </location>
        <location id="id2">
            <name>Off</name>
        </location>
        <init ref="id2"/>
        <transition>
            <source ref="id1"/>
            <target ref="id0"/>
            <label kind="synchronisation">low!</label>
            <label kind="assignment">reset()</label>
        </transition>
        <!-- missing transitions -->
    </template>
    <!-- missing templates for Lamp and User-->
    <system> system Lamp, LampController, User;</system>
</nta>
```

Figure A.2: Excerpt of the XML based DSML for the Lamp example.

system is going to be initialized. In our lamp example, we have one instance of each of the *User*, *Lamp* and *LampController* automata. In general however, a system might contain multiple instances (e.g. multiple users) of a single automaton.

A.3 Executable Model Generation

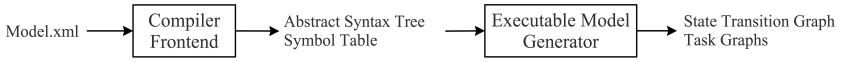


Figure A.3: Overview of the executable model generation.

A network of timed automata as described in Section A.2 is a system model with sufficient detail to be interpreted. The model interpretation can be divided into two steps: 1) Executable Model Generation, and 2) Model Execution. Both are handled in sequence each time a model is executed. In this section we present the model generation and in Section A.4 we present the model execution.

An overview of the executable model generation is presented in Figure A.3. The input is an XML based DSML describing the system (`Model.xml`), the final result (`State Transition Graphs`, `Task Graphs`) is an internal representation of the system that later will be executed by our virtual machine. The executable model generation is divided into two steps: 1) A *compiler frontend* that parses the input XML file and creates a single abstract syntax tree (AST) and a symbol table. 2) An *executable model generator* that traverses the AST to generate the final executable model representation.

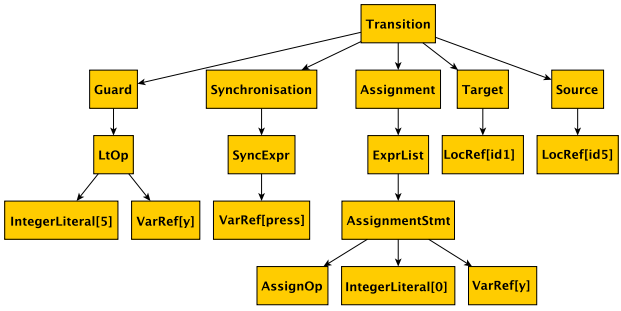


Figure A.4: AST for transition Low to $TurningBright$.

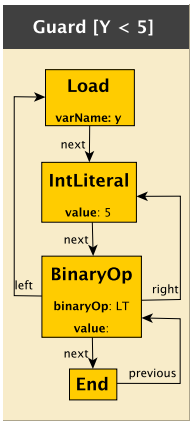


Figure A.5: Task graph for guard y < 5

The compiler frontend uses standard compiler techniques and will not be described in detail. In short, UPPAAL DSML is defined as a context-free grammar that can easily be used to generate a parser using the Antlr [152] parser generator tool. The resulting AST is then traversed once more to construct a symbol table, a mapping from scopes to variable declarations. A scope in the UPPAAL DSML can be a global declaration, system declaration, template declaration, or a function.

Figure A.4 shows a subtree of the AST representing the transition Low to $TurningBright$ in LampController. Apart from source and target information of the transition (`Source`, `Target` subtrees) it also includes three labels: `Guard` ($y < 5$), `Synchronization` ($press?$), and `Assignment` ($y = 0$).

The executable model representation later to be executed by a virtual machine consists of two parts: 1) *State transition graphs*, one for each automata, and 2) a number of *Task graphs*. The state transition graphs are just an internal graph representation of the system's timed automata as described in Section A.2. There exists one graph for each automaton. The states are nodes and the transitions are edges. Both nodes and edges are annotated with references to task graphs. Each transition label (guard, synchronization, update) is represented by a separate task graph, and each node attribute (invariant) is also represented as a task graph.

A task graph defines the control flow of a task graph evaluation. It consists of a collection of task nodes that are connected with *next* and *previous* attributes. Each task node has a task type attribute defining the role of that node. Examples of task types are: *DECL* declares a variable, *LITERAL* defines a integer literal, *BINARY_OP* for binary operations, *STORE*/*LOAD* store/load a variable value from/to the heap, *END* signals the termination of a task graph evaluation, etc. Depending upon task type a node can have additional attributes, e.g. the task node for binary operators have *left* and *right* attributes pointing to left and right expression nodes.

Fig. A.5 shows the task graph for the guard $y < 5$ of the Low to $TurningBright$ transition in Lamp Controller. The execution order is defined by the *next* edges (non-essential *previous* edges are omitted for simplicity) and the less-then operator is represented as a binary operation (tagged with LT) with two node type specific edges (*left* and *right*) referencing the values to be used in the operator. The *previous* edge in the *END* node points to the final result of the task graph evaluation.

In addition to task graphs generated due to various state and transition attributes we also generate task graphs for all declarations of global variables and signals defined in the `<declaration>` part of the AST, and for all clock and variable declarations local to a certain automaton. These additional task graphs are not directly referenced by any transition graph, they will be used in the initialization phase of the virtual machine before the execution starts.

Task and transition graphs are generated in a single AST traversal. Due to space limitations the actually used algorithm will not be presented here.

A.4 Model Execution

The model interpretation starts with an initialization phase (Section A.4.1) where global and template variables are declared and initialized, the real-time time unit is set, and connections to environment models are established. Then the actual execution can start (Section A.4.2).

The core of the model interpreter is the *timed automata virtual machine* (TAVM). Apart from heap and stack management the TAVM has two parts that together are responsible for the actual execution. The *state transition machine* (STM) is responsible for the state transitions, and the *task graph interpreter* (TGI) is (on requests from the STM) evaluating task graphs. Several of the design decisions for the TAVM are inspired by UPPAAL Tron [103], a model based testing tool from UPPAAL.

A.4.1 Virtual Machine Setup

A.4.1.1 Declarations:

The first step is to execute global and system declarations by the task graph interpreter in order to initialize all variable and clock declarations used by the system. For example, it declares what channels are going to be used. The system declarations provide a list of automata instances that are to be executed by the virtual machine. Finally, for each instantiated automaton, all local declarations are executed and a list of initial active states is created.

A.4.1.2 Model time unit:

In timed automata, a time tick is an abstract entity that can be assigned to any real time unit, e.g. milliseconds, seconds, minutes, etc. In order to correctly behave as real-time clocks, the TAVM must know the real time unit of a tick. It therefore provides a method $setRealTimeUnit(milliseconds)$ to set the time unit in milliseconds. How clocks progress is discussed in more detail at Section A.4.2.

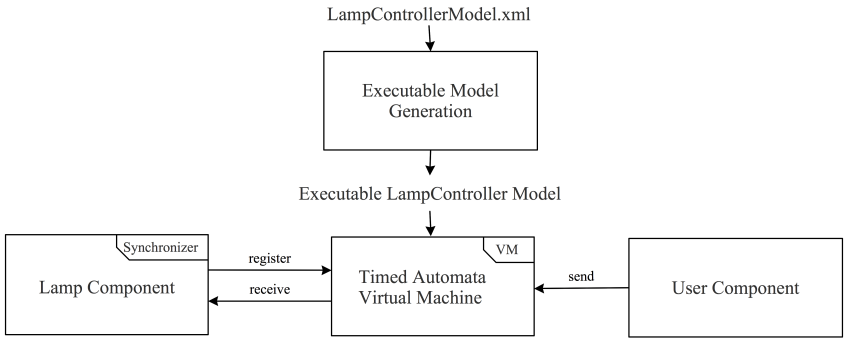


Figure A.6: Overview of the Lamp example interpretation.

A.4.1.3 Environment connection:

As mentioned in Section A.2.2, the entire model is divided into two categories: Environment models representing external components that interact with the running system, and system models that are to be executed in the TAVM. The TAVM connects with the environment through signals defined in the automata model.

Figure A.6 shows an overview of the Lamp example interpretation. The input is an XML based system specification (LampControllerModel.xml) that is used to generate an executable model which is then fed to the TAVM for execution. At runtime, TAVM must be connected to a real lamp and a real button. To realize this in our approach, we replace the models of the environment with an actual environment represented by the Lamp and User components in Figure A.6, and the TAVM executes only the Lamp Controller model. A component in this case is a piece of software which handles the communication with external devices.

The TAVM assigns a unique identifier to each channel. This identifier can be used to send and receive signals from the TAVM. TAVM has a public interface (named VM) providing a method $getChannelId(\text{“}channel\text{”})$ that can be used to get channel identifiers. To send a signal, the VM interface provides a $send(channelId)$ method that can be used to send a signal from the environment to the virtual machine. Data can also be sent using the $send$ method as a string expression like “$a = 2$”. These expressions are converted to task graphs on-the-fly and evaluated by the task graph interpreter when a signal is consumed. More details about how signals are consumed are provided in the next section.

The TAVM provides an abstract class *Synchronizer*, that should be extended by components interested in receiving signals from the TAVM. A component registers itself for a certain channel by, first, getting the channel identifier using the $getChannelId$ method, and then call the $register$ method provided by the VM interface. The $register$ method take three parameters: 1) a channel identifier identifying which type of signal we are interested in, 2) an instantiation of the $Synchronizer$ class, that will receive the signals, 3) and a array of variable names specifying what variable values we are interested in. The $Synchronizer$ class defines one abstract method $receive$ that has two parameters: 1) a channel identifier

that can be used to determine which signal is received, 2) the data that comes with the signal.

A.4.2 Virtual Machine Execution

The TAVM provides a $start$ method which starts the actual execution once the setup is completed. Once started, the virtual machine is idle until triggered either by input from the environment, or by a time tick. The heart of the TAVM is the State Transition Machine (STM). The STM keeps track of all active nodes and decides what and when transitions are triggered. The STM is using another component, the Task Graph Interpreter (TGI), whenever a task graphs needs to be evaluated. In what follows we first present the STM and then the TGI.

A.4.2.1 The State Transition Machine.

The STM maintains a set of all active nodes N and a set S, representing the current state, containing N and the values of all the variables and clocks. From now on "state" refers to the global state S and we refer to individual timed automata states/locations as nodes. Upon start, the STM checks all instantiated models and execute those that are in a committed state. To do that, the STM checks (one by one) all the active nodes in N, if a node is in a committed state, then STM randomly selects one outgoing transition from that node and tries to execute it. If that transition cannot be taken (e.g. a guard evaluates to false), STM tries another one. This process is repeated until all committed nodes are handled, and will also be repeated after each taken transition ending up in a committed state. The STM supports non-determinism by randomly selecting nodes and transitions if multiple available.

Algorithm 1 shows the pseudo code for executing one transition which does not interact with the environment. That is, it can only handle signals sent and received within the system model. The handling of signals involving external components is discussed later on.

In what follows, S' is a temporary state that can be rolled back to S, or S can become S', and if there is no guard on a transition (or invariant on a node), then the evaluation of the guard (invariant) expression returns true. Evaluation calls (e.g. $evaluateGuard(transition, S)$) are calls to the task graph interpreter requesting a task graph guard ($transition$) to be evaluated in a given state (S).

Algorithm 1 starts by making sure that a transition can only be taken when the guard of the transition is true (line 2). If guard is true, and the transition involves synchronization (line 3), then it makes sure that the guard of the receiving transition is also true (line 6). After these preliminary checks we have a potential transition to a new state and we clone the current state ($S' \leftarrow S$, line 10) to make sure that we can roll back to S if future steps fails. Then we start to evaluate the update task graphs (line 11, 13), and verify that all invariants still holds (line 15). These steps might update S' and still fail. If they succeed we decide to make the transition and update the current state $S \leftarrow S'$ (line 16) and update N by adding and removing the old and new active node (also for the signal receiving transition), lines 17-22.

A1 Algorithm for executing a transition

Input N set of all active nodes
Input S current state including N and the value of all variables and clocks
Input $transition$ to be taken and it's source $node$
Return $true$ if transition accepted, otherwise $false$

```
1.  recvTransition ← NULL
2.  if evaluateGuard(transition, S) == true then
3.    if transition.synch ! = null and transition.synch.type == SEND
      then
4.      channelId = evaluateSynchronization(transition, S)
5.      recvTransition = findReceivingTransition(channelId, N, S)
6.      if evaluateGuard(recvTransition) ! = true then
7.        return false
8.      end if
9.    end if
10.   S' ← S
11.   evaluateUpdate(transition, S')
12.   if recvTransition! = null then
13.     evaluateUpdate(recvTransition, S')
14.   end if
15.   if checkAllInvariants(N, S') == true then
16.     S ← S'
17.     N.remove(node)
18.     N.add(transition.targetNode)
19.     if recvTransition ! = null then
20.       N.remove(recvTransition.srcNode)
21.       N.add(transition.targetNode)
22.     end if
23.     return true
24.   else
25.     discard(S')
26.     return false
27.   end if
28. end if
29. return false
```

In order to communicate with the environment, we must modify our algorithm at a few places. For sending a signal to the environment, and after getting $channelId$ of the sender, we must look at the list of registered synchronizers. If any synchronizer is found registered for the same channel, we take the transition after executing update task graph and evaluating all the invariants. Then we call the $receive$ method of the associated instance of the $Synchronizer$ class with the requested data.

When a signal is received from the environment, the STM finds the receiving transition through $channelId$, and execute the *guard* and *update* task graphs. It

can happen that the system models in the STM and the environment models are not synchronized, and there is no transition at the moment who could receive the signal. STM then takes a flexible approach, and if the signal is not consumed, that signal is moved to a queue. Then the queue of signals is checked repeatedly whenever the clock ticks or a new signal arrives to consume the pending signals.

The STM maintains an internal timer, whose time period can be configured as discussed in Section A.4.1. The STM keeps an internal data structure for all the clocks in the model. When the timer ticks, the STM temporarily increases the time of all the clock variables modelled in the automata, i.e, S' and checks the invariants of all enabled nodes. If all the invariants hold, the STM increases time for all the clock variables permanently $S \leftarrow S'$ and the timer goes to wait state. If the invariants of any active nodes are violated by the temporary increment of the timer, the STM reverts the time increment S and executes those nodes first, whose invariants are violated, by evaluating their transitions as discussed in the Algorithm 1. If a node can not take a transition, then the system ends in a timelock (this points to a design flaw in the model). The STM will then stop execution and throw a $TimelockException$.

If the selected time tick unit is very small we might end up in a situation where the TAVM can not manage all the required computations (or transitions) before the next tick. In addition to the general STM overhead this might occur when waiting for an external signal or due to certain time consuming TGI computations to check if a transition is possible or not. Our implementation handles this situation by buffering the time ticks and then executes them as soon as possible. This is (of course) problematic since it might cause a delay in the signals sent to the real world components. Thus, for each application, the real time unit to be used should be chosen carefully to make sure that the TAVM always manage to do all the required work before the next tick.

A.4.2.2 The Task Graph Interpreter.

The task graph interpreter (TGI) evaluates task graphs on request from the STM. On initialization of the model, the STM requires the TGI to evaluate the initialization expressions for all the declared variables. Later on the TGI evaluates the *guard* and other transition and state attributes to take transitions as described previously in Algorithm 1. The TGI keeps track of a heap which stores all the global, system and template declarations, and a stack to store the state of local variables and function parameters when a call occur. Algorithm 2 shows an excerpt of the algorithm used by the TGI. With each evaluation request the STM also provides the $processId$ that is needed to know which variables belongs to which instantiated model. For evaluating global and system declarations, the STM uses 0, and -1 respectively as $processId$. The CT (Current Task) always points to the current task. Upon receiving a request for evaluation, the TGI checks the task type (line 3, 6, 10) and takes the appropriate action. Once a task is evaluated, CT moves to the next task (line 20). This process is repeated until CT reaches the END task, which stops the task graph evaluation and the result of the evaluation is returned.

A2 Algorithm for task graph interpretation

Input $taskGraph$ to be executed and the model instance identifier $processId$
Return Result of the task graph evaluation

```
1.  CT ← taskGraph.getFirst()
2.  while CT ∉ END do
3.    if CT ∈ LOAD then
4.      varName ← CT.getVarName()
5.      CT.value = heap.get(processId).get(varName)
6.    else if CT ∈ STORE then
7.      varName ← CT.getVarName()
8.      value ← CT.getPrev().getvalue()
9.      heap.get(processId).get(varName).setValue(value)
10.   else if CT ∈ BINARY_OP then
11.     op ← CT.getOp()
12.     if op ∈ LT then
13.       CT.value = CT.getLeft().getValue() <
          CT.getRight().getValue()
14.     else
15.       ... more operators here
16.     end if
17.   else
18.     ... more tasks here
19.   end if
20.   CT ← CT.getNext()
21. end while
22. return CT.getPrev().getvalue()
```

A.4.3 Validation

Apart from extensive in-house testing, our model interpreter has been evaluated in various adaptive systems. The original idea was presented in [109] where the adaptation logic of a robotic system is formally verified and executed by the model interpreter. Later on the model interpreter was evaluated in several case studies, including a smart house system, a security system, and two vehicular traffic systems [79]. Other applications where we applied the model interpreter are a digital story telling application and an e-health system [170]. See the project website [1] for more details about these case studies.

A.5 Additional Features and Future Work

Direct access to the model at runtime provides many additional advantages. Some of these features are already implemented and tested (Section A.5.1) whereas others can be considered as future work (Section A.5.2). See [109] for more details.

A.5.1 Additional Features

System Model Updates. The model interpreter supports online updates of the system models, which is crucial to deal with bugs, or adding new functionality to the running system. Our approach follows the classical process of runtime updates based on quiescence states [123]. The model interpreter provides a method $changeModel(model)$ which receives an updated model description (DSML). After that, the interpreter waits until each automaton of the current model reaches a quiescence state (i.e., no ongoing input or time triggered transactions) and interrupts the execution. The state of the current model is then saved and any new external inputs received while the update takes place are stored in a buffer. The interpreter then generates a new executable model (Section A.3) and initialize that model (Section A.4.1). Next, the interpreter restores the saved state of the previous model to the updated model and initializes new variables if applicable. Finally, the TAVM restarts the execution using the updated model.

Goal Verification. The model interpreter provides basic support for runtime verification of system goals. The *goal manager* component in the interpreter provides a function $addGoal(goal, client)$ that register goals to be monitored. A goal is a boolean expression involving clocks and variables (e.g. $y \leq 10$). The client is an implementation of the $GoalClient$ interface registering to receive updates of the goal status. When a goal is registered the interpreter converts it to a task graph and start to notify the client every time a goal status is changed. Using this approach an interested component can track state changes and check whether the system goals hold or are violated. This feature was used in [109] to verify the correctness when updating the feedback loop models to deal with a new set of adaptation goals in a self-adaptive system.

Model Visualization. The model interpreter also comes with a graphical user interface allowing a user to inspect the running model, its ongoing execution, and to monitor variable values. This is useful for debugging the running system. The model interpreter provides a probe for interested components to get updates of the running model. The goal manager used in the goal verification uses the probe to listen to the updates and notifies the graphical user interface which display the current status of the model, see, e.g., Fig. 13-15 in [109].

A.5.2 Future Features

The goal manager currently used for both goal verification and model visualization has certain limitations. For example, goal types are limited to only boolean expressions. In the future we plan to provide an interface offering plug and play facilities for arbitrary external components, and this new interface should give access to the complete model of the system (including the environment models) and allow every type of expression that can be represented as a task graph to be evaluated. This new machine interface opens up the possibility for a wide range of components to be attached to the virtual machine.

Our primary candidate for such plugin component is *online verification*. UPPAAL is foremost an offline verification tool. Given a model and a set of TCTL properties, the tool can prove that these properties are never invalidated. However, due to the so-called state explosion problem, incomplete knowledge about environment and memory constraints, offline verification may not be achieved. The interpreter on the other hand has runtime access to the complete model and can after each transition verify that the provided TCTL properties, converted to task graphs, are still valid. It is not a formal verification, it is however a pragmatic approach to verify that the running system behaves correctly. We are currently implementing an online verification component providing support for a subset of the timed computation tree logic (TCTL) properties, like constraints, safety and liveness properties.

Another possible approach to model checking problem is to delegate that work to other model checking tools. For example, using the plugin mechanism the model interpreter should be able to incorporate other trusted external modules (e.g., runtime model checking engines to support continuous verification at runtime).

A.6 Related Work

Ever since D.C. Schmidt's seminal paper on Model-Driven Software Engineering in 2006 [167] the interest for various aspects of model-driven design has flourished. In our approach we take the model-centric approach one step further and consider the model not only as a vehicle for code-generation, but also as a design specification suitable for verification. The number of existing models (DSMLs) that can be verified, executed in a real world environment, and that allows runtime model updates are rather few.

The Foundational Subset of Executable UML (fUML) defines the semantics for a subset of UML that can be executed by the fUML execution engine [143]. The fUML execution engine executes an in-memory representation of fUML models. Progress in the verification of these models has recently been achieved [128] but, to the best of our knowledge, no progress has been made yet for runtime model updates.

Ghezzi et al. [91] introduce adaptive model-driven execution to mitigate non-functional uncertainties. Using UML interaction diagrams a Markov decision model of the system is generated. The model is augmented with probability distribution of different execution paths of the system. The model is then executed by an ad-hoc interpreter that drives the execution of the system according to specified probabilities to guarantee the highest utility for a set of quality properties. In their model each state is associated with an implementation of an abstract functionality of the system, and the interpreter invokes the implementations while state-by-state traversing the automaton, whereas we model and execute the actual implementation of the system. Markov decision models are well-known to allow probabilistic model checking and verification tools are available [126].

Anlauf et al. [5] presents an interpretable language XASM (Extensible Abstract State Machine). XASM uses a notion of external functions as defined in ASMs to realize a component-based modularization. The support environment of XASM consists of the XASM-compiler translating XASM programs to C source code, the runtime system, and the graphical debugging and animation tool. This approach lacks support for runtime update of the model, and although computer-aided verification of ASM models is possible in theory, it is well-known to be difficult in practice [176].

A.7 Summary and Conclusions

In this paper, we presented a model interpreter for timed automata, a formalism often used for modeling and verification of real-time systems. In addition to handling real-time features, it is the use of a domain specific model being verifiable, executable in a real world scenario, and allowing model updates at runtime that makes our approach rather unique. Given a model of the system the interpreter converts it into an executable model that can be interpreted by a timed automata virtual machine. Contrary to traditional approaches, where models are converted to code, using a model interpreter provides a number of additional advantages: 1) models are executed directly without converting them to a source code; hence no model-based testing is required, 2) models can be replaced at runtime without stopping the system, e.g., to add new functionality, 3) models can be used to verify system properties at runtime, 4) and it is also possible to visualize the running models. Our virtual machine can handle real-time system features like simultaneous execution, system wide signals, a ticking clock, and time constraints, not usually handled by ordinary stack based virtual machines. We included a future work section pointing out the possibility to use a model of the entire system to perform online verification.

A byte code version of the model interpreter can be downloaded from the project website [1].

Appendix B

DeltaIoT: A Self-Adaptive Internet of Things Exemplar

Abstract

Internet of Things (IoT) consists of networked tiny embedded computers (motes) that are capable of monitoring and controlling the physical world. Examples range from building security monitoring to smart factories. A central problem of IoT is minimising the energy consumption of the motes, while guaranteeing high packet delivery performance, regardless of uncertainties such as sudden changes in traffic load and communication interference. Traditionally, to deal with uncertainties the network settings are either hand-tuned or over-provisioned, resulting in continuous network maintenance or inefficiencies. Enhancing the IoT network with self-adaptation can automate these tasks. This paper presents DeltaIoT, an exemplar that enables researchers to evaluate and compare new methods, techniques and tools for self-adaptation in IoT. DeltaIoT is the first exemplar for research on self-adaptation that provides both a simulator for offline experimentation and a physical setup that can be accessed remotely for real-world experimentation.

B.1 Introduction

Internet of Things (IoT) are composed of tiny embedded computers (motes) equipped with low-power wireless networking, sensors and actuators. These motes form networks that are capable of monitoring and controlling the physical world and thereby connecting digital processes to our physical environment. IoT are expected to have a broad impact across diverse domains such as manufacturing [218], farming [119] and monitoring of remote medical facilities [159]. Due to size and cost concerns, IoT motes typically offer limited computation, storage and energy resources. This demands careful design of the IoT network. Realising efficient IoT systems is particularly important as wireless communication consumes the majority of energy for a typical IoT device [2].

IoT network technologies are growing in both maturity and capability [156]. The state-of-the-art offers both short-range multi-hop networks with very high reliability [66] and single-hop networks that offer a range of several kilometers [159]. Through careful configuration of the network settings both approaches can achieve

multi-year battery lifetimes and high levels of reliability. However, finding the right network settings is hard as IoT applications are subject to a variety of uncertainties, such as sudden changes in traffic load and communication interference. Traditionally, the network settings are either hand-tuned or over-provisioned to deal with uncertainties, resulting in continuous network maintenance or inefficiencies.

Self-adaptation [48, 134, 203] provides the means to automate these tasks. To that end, a feedback loop can be deployed on top of the network to monitor and assess the motes and the environment to autonomously adapt the IoT system. Examples of research that apply dynamic adaptation of IoT networks are [18, 45, 136, 169, 190, 202]. Exemplars enabling researchers to evaluate and compare new self-adaptation solutions have been proposed for different domains, e.g. for client-server systems [50], cloud environments [14], service-based systems [204], and cyber-physical systems [120]. However, there is no exemplar available to support researchers in IoT.

This paper presents an IoT exemplar to support research on self-adaptation in IoT. The exemplar applies multi-hop communication in IoT, where each IoT mote must have a path towards the gateway along other motes. Our motes use LoRa radio technology supporting long range communication.[1] The focus is on dynamically adapting the settings of the IoT network under various types of uncertainties. The examples in this paper apply architecture-based adaptation, realised by means of a Monitor-Analyse-Plan-Execute-Knowledge feedback loop (MAPE-K) [65, 116, 211]. However, the exemplar does not prescribe a particular type of self-adaptation realisation.

The DeltaIoT exemplar comprises a simulator for offline experimentation and a physical setup of 25 motes that can be accessed remotely for experimentation in the field. The IoT system is deployed at the Campus of the Computer Science Department of KU Leuven. DeltaIoT is the first exemplar for research on self-adaptation that provides both a simulator and a physical setup for experimentation. Detailed information of the DeltaIoT exemplar is available via the exemplar website.[2]

The remainder of this paper is structured as follows. In Section B.2, we introduce the DeltaIoT exemplar and provide scenarios for comparing self-adaptation solutions. Section B.3 explains the architecture of both the physical IoT network and the simulator. We also present the unified interface that supports self-adaptation of the IoT system. In Section B.4, we explain how to set up an experiment with DeltaIoT and illustrate how adaptation can be realised using a simple example. Finally, we draw conclusions in Section B.5.

B.2 DeltaIoT Exemplar and Adaptation Scenarios

The DeltaIoT exemplar is part of the smart campus initiative by DistriNet, KU Leuven Belgium. DeltaIoT consists of a multihop network based on LORA com-

[1] https://www.lora-alliance.org/What-Is-LoRa/Technology
[2] https://people.cs.kuleuven.be/danny.weyns/software/DeltaIoT/

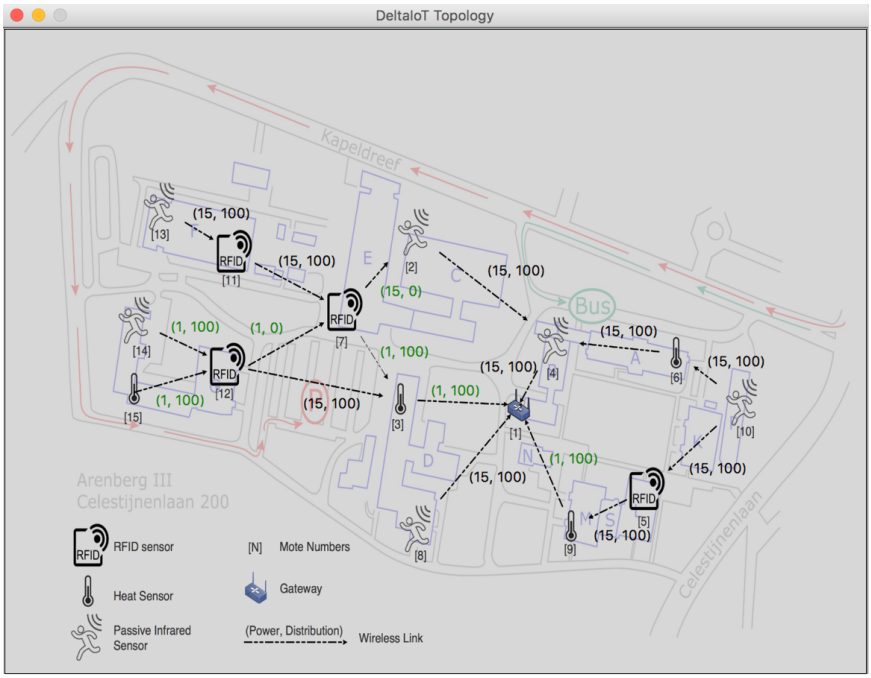


Figure B.1: DeltaIoT network topology

munication comprising 25 motes that are distributed in various buildings of the Department of Computer Science at KU Leuven. Figure B.1 shows a subset of 15 of the IoT motes deployed in various buildings at the campus.

In each building, motes are strategically placed to provide access control to labs (via RFID sensor), to monitor the occupancy status (via passive infrared sensor) and to sense the temperature (via temperature sensor). The sensor data from all the buildings are relayed to the IoT gateway, which is deployed at a central monitoring facility. Campus security personnel monitor the status of various buildings and labs from the monitoring facility, and take appropriate action whenever the unusual behaviour is detected in the buildings.

As shown in Figure B.1, each IoT mote in the network relays its sensor data to the gateway. Some of the IoT motes are not within the direct reach of the central gateway and have to relay their sensor data via intermediate motes. This method of communication is called multi-hop communication. In multi-hop communication, each IoT mote must have a path towards the gateway. To that end, each IoT mote in the network interacts with other motes to form paths towards the gateway. This process requires the configuration of each IoT mote. Radio communication typically dominates the energy consumption of IoT motes. However, IoT applications are expected to last longer on a single battery, while offering reliable communication. It is therefore important to *reduce the energy consumption* of the IoT motes, while *guaranteeing high packet delivery performance*.

For reliable and efficient communication, the IoT motes should be optimally configured. However, the packet delivery performance and energy consumption of IoT motes are influenced by a number of uncertainties leading to different adaptation scenarios for DeltaIoT, which are presented in Table B.1. These generic scenarios are organised by type of uncertainty that makes self-adaptation necessary, type(s) of adaptation required, and type(s) of goals that these adaptations aim to meet. Within these scenarios, we propose the evaluation and comparison of different self-adaptation solutions based on quality attributes and metrics that are summarised in Table B.2.

Scenario 1 considers interference of the wireless network. The aim is to reduce packet loss and minimise energy consumption by adapting the transmission power of motes (higher power will results in less errors but increase energy consumption) and/or adapt the proportion of traffic send by the motes to their respective parents (select links with lower levels of messages lost, taking into account the energy consumed). Scenario 2 considers fluctuations in the traffic generated by motes (or their children) in addition to interference of the network. The goals and the types of adaptation are the same as for scenario 1. Scenario 3 considers the same types of uncertainties as scenario 2, however the adaptation goals are to minimise both packet loss and energy consumption. This scenario offers the modification of the spreading factor as an additional type of adaptation. The spreading factor is a measure for the number of bits encoded per symbol of a transmitted packet. A higher spreading factor results in longer communication range and more redundant data but at the cost of a reduced payload of the message and an increase of the time on air requiring more energy [159]. Scenario 4 considers motes that are lost (temporary or permanently). To minimise the packet loss, the paths to the gateway can be adapted. Scenario 5 considers the challenging problem of mobile motes in an IoT setting. To minimise packet loss and maintain connectivity, the motes can dynamically add and remove links. Finally, scenario 6 considers a decentralised setting where multiple gateways manage traffic in subnets of the overall network. The particular focus of the scenario is on deciding the traffic routed to the different gateways (for example by the motes at the borders of the subnets). To minimise packet loss and energy consumption, and balance the energy consumption in the subnets, the gateways have to agree on the traffic routed to each of the gateways.

To evaluate and compare solutions the following parameters can be measured: the number of packets sent and received (for reducing packet loss), the energy consumption of individual motes and the whole network (for minimising energy consumption; in Coulomb), the smallest number of lost packages (for minimising packet loss), the links added and removed (for maintaining connectivity while minimising packet loss), and the energy consumption in subnets managed by the different gateways (for balancing energy consumption).

B.3 Architecture of DeltaIoT

We explain now the architecture of DeltaIoT and the interface that supports probing and effecting the IoT system. We also explain how adaptation can be realised.

Table B.1: Generic adaptation scenarios for DeltaIoT.

Scenario	Type of uncertainty	Type(s) of adaptation	Type(s) of adaptation goals
S1	Wireless interference	Adapt the transmission power and/or modify the path to the gateway	Reduce packet loss and minimise energy consumption
S2	Wireless interference and fluctuation traffic load	Adapt the transmission power and/or modify the path to the gateway	Reduce packet loss and minimise energy consumption
S3	Wireless interference and fluctuation traffic load	Adapt the transmission power and/or modify the spreading factor settings and/or modify the communication paths	Minimize packet loss and minimise energy consumption
S4	Mote lost (temporary or permanently)	Modify the path to reduce the impact of the lost mote	Minimise packet loss
S5	Mobile motes	Dynamically add and remove links	Maintain connectivity and minimise packet loss
S6	Partial knowledge in subnets managed by multiple gateways	Dynamically decide the traffic routed to the different gateways	Minimise packet loss, minimise energy consumption, and balance energy consumed in subnets

Table B.2: Quality attributes and metrics for the evaluation and comparison of self-adaptation solutions to DeltaIoT.

Quality attribute	Metric
Reduce packet loss	Bounded number of packets lost in the network
Minimise energy consumption	Energy consumption of the entire network
Minimise packet loss	Smallest possible number of packet lost
Maintain connectivity	New links added and old links removed
Balance energy consumption	Energy consumption in subnets managed by gateways

Finally, we give an overview of the architecture of the DeltaIoT simulator that provides a compatible interface to realise self-adaptation.

B.3.1 Architecture of DeltaIoT Deployed at KU Leuven

Figure B.2 shows the architecture of the DeltaIoT system that consists of four tiers: network tier, gateway tier, management tier, and managing system tier. We briefly explain the role of each tier, from bottom to top.

B.3.1.1 IoT Network Tier

The IoT network tier consists of the IoT motes connected via a wireless communication network. Figure B.1 shows an example topology of the network.

B.3.1.2 Gateway Tier

The gateway tier is the root for the IoT network. All the IoT motes in the network transmit their data to the gateway, which is then consumed by the users based on their application requirements. Thus, all the IoT motes must have a route towards the gateway.

B.3.1.3 Management Tier

The IoT motes and the gateway can be monitored and managed via the management tier. The management tier consists of three key building blocks:

Webservice Engine: connects DeltaIoT to the external world via Internet through a web service. This service enables an external entity to query the status of the network and modify the network settings. We explain the format of interactions with the Webservice engine below.

Statistics Engine: is responsible for collecting and storing network statistics. All the network related information is maintained in a database. This building block provides information about the packet delivery performance and the energy consumption of all the IoT motes in the network.

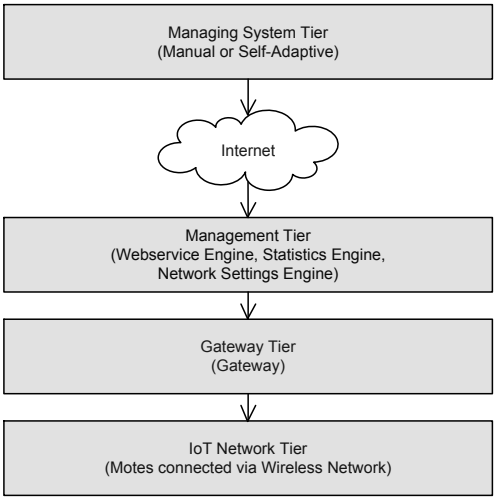


Figure B.2: Architecture of DeltaIoT Deployed at KU Leuven

Network Settings Engine: interacts with the IoT network via the gateway to collect data and adapt the network settings of the IoT motes. We explain the format of interactions with the IoT network below.

B.3.1.4 Managing System Tier

The managing system tier contains the entity that is responsible for managing the IoT network via the management tier. In a traditional setting, network management is done manually by a system administrator. In a self-adaptive setting, the management tier consists of a feedback loop that automatically deals with adaptation of the network settings based on a set of adaptation goals. The exemplar offers a Java client to access the web service provided by the Webservice engine.

B.3.2 Interface to Realise Self-Adaptation

The Webservice Engine provides an interface for probing and effecting the IoT system. This interface is defined in a WSDL file.[1] Access to the web service is regulated, as only one entity is allowed to perform self-adaptation at a time. The exemplar website provides details how users can register and access the web service. We first give an overview of the web service interface. Then we explain how adaptation of the IoT network can be realised.

As explained above, the exemplar offers a client that comprises a Java package with Probe and Effector classes. The client hides the details of the interaction with the web service. Listing B.1 lists the methods of the probe that can be used to monitor the IoT network.

Listing B.1: DeltaIoT probing methods.

```
ArrayList<Mote> getAllMotes();
```

[1] This file provides a description of the operations and messages of the web service in the Web Services Description Language.

```
double getMoteTrafficLoad(MoteId);
double getMoteEnergyLevel(MoteId);
int getLinkPowerSetting(Source, Destination);
int getLinkSpreadingFactor(Source, Destination);
double getLinkSignalNoise(Source, Destination);
int getLinkDistributionFactor(Source, Destination);
ArrayList<QoS> getNetworkQoS(Period);
```

The method $getAllMotes$ returns an array with a representation of all the motes of the network. The methods $getMoteTrafficLoad$ and $getMoteEnergyLevel$ return the traffic generated by a mote and the energy consumed for that traffic respectively. The method $getLinkPowerSetting$ returns the setting of the transmission power that a source mote uses to communicate with a parent (destination mote). A higher power setting results in lower packet loss but at the cost of more energy consumption. Similarly, the method $getLinkSpreadingFactor$ returns the value of the spreading factor for a link. Recall that a higher spreading factor results in longer range but at the cost of a reduced payload and more energy consumption. The method $getLinkSignalNoise$ returns the signal to noise ratio (SNR) for a link between a source and destination mote. SNR represents the ratio between the level of the desired signal and the level of the noise, which comes from the environment in which the IoT system operates. SNR provides information about the quality of the link by estimating the ratio between the interfering RF signal and the data RF signal. The lower the SNR, the higher the interference, resulting in higher packet loss. Finally, the method $getLinkDistributionFactor$ returns the percentages of the messages sent by a source mote over a link to one of its parents. The total sum of the distribution factors for one mote is normally 100.[2] The last method, $getNetworkQoS$ returns statistical data about the quality of service (QoS) of the overall network for a given period. Currently this method returns data about packet loss and energy consumption of the network.

Listing B.2 lists the methods that can be used to adapt the IoT network, i.e., adapt the network settings of the motes.

Listing B.2: DeltaIoT effecting methods.

```
void setMoteSettings(MoteID, List<LinkSetting>);
void resetDefaultConfiguration();
```

The method $setMoteSettings$ allows to set the parameters for the parent links of a mote with a given ID. A $LinkSetting$ contains the source and destination node of the link, the transmission power and spreading factor to be used to communicate via the link, and the distribution factor for the link. Finally, the method $resetDefaultConfiguration$ resets the network settings to the original values. This method can be used to bring the system to a well-known state, e.g. as failsafe state.

B.3.3 Adaptation of the DeltaIoT Network

Adaptation of the IoT network is based on the adapting the network settings of the motes that participate in the IoT network. We explain adaptation with the

[2] If packets are duplicated and sent to more than one parent, the sum of the distribution factors will be above 100.

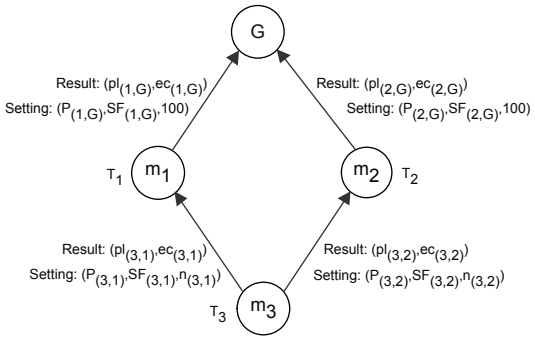


Figure B.3: Excerpt topology with a network setting and result

excerpt subnet shown in Figure B.3. Each link between a pair of motes connects a source node with a destination node. The radio transmission by the source node is received by a destination node. In this context, we assume that a communication is unicast, where each transmission from a source node reaches a single destination node.[3] Furthermore, the communication between the source and destination is successful only if the transmitted packet from the source reaches the destination node. Unsuccessful transmission implies a packet loss.

A mote m_i produces traffic T_i. Each mote sends both the traffic it produces and the traffic it receives from its children to one of its parents. The gateway collects all traffic for the user. The network setting for a mote s to communicate a packet over a link to a parent d (i.e. the destination) is defined as a tuple: $(P_{(s,d)}, SF_{(s,d)}, n_{(s,d)})$, where $P_{(s,d)}$ is the transmission power used for the communication over the link, $SF_{s,d}$ is the setting for the spreading factor used for the communication,and $n_{(s,d)}$ is the distribution factor, i.e., the percentage of the traffic that is sent by mote s to parent d. As explained above, the sum of the distribution factors for one mote is normally 100.

For each link in the network, we offer external entities the possibility of changing the settings for $P_{(s,d)}$, $SF_{(s,d)}$ and $n_{(s,d)}$. The power setting $P_{(s,d)}$ can be set between -3 and 18 to get a required SNR for the communication to a parent. The spreading factor $SF_{(s,d)}$ can be set to a value in a range from 7 and 12. The spreading factor can be selected per link or it can be fixed for the mote or the network as a whole. Figure B.4 shows the relationship between the power settings and the SNR for one of the links in the network with a spreading factor set to 11. The data of this example is based on experimental data from observations in the field. The distribution factor $n_{(s,d)}$ can be set from 0 to 100 in steps of 10. From Figure B.3, we can see that some motes in the network may have more than one parent link. In such cases, the packets from a source mote can reach the gateway via multiple links. It is therefore important to select links for each mote so that the packets from the source mote reaches the gateway successfully. This link selection process involves analysing the characteristics of each link. Some links may have high packet loss compared to other links. The packet delivery performance of a

[3] This assumption is based on time-synchronised scheduling of message communication [66].

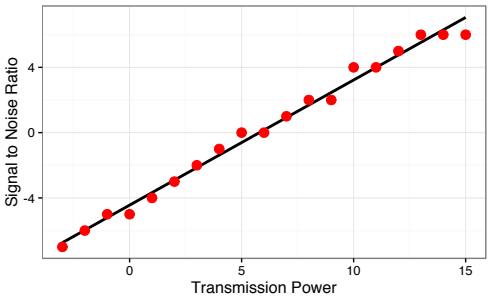


Figure B.4: Transmission power to SNR relation (for Link between Mote11 and Mote1 with a spreading factor of 11)

link depends on the characteristics of the operational environment and the wireless interference [223].

The power settings of the motes along with the spreading factors and the selection of paths determine the energy consumption and the packet loss of the IoT network. To determine the network performance, the management tier collects data about the packet loss $pl_{(s,d)}$ and the energy consumption $ec_{(s,d)}$ for a given network settings $(P_{(s,d)}, SF_{(s,d)}, n_{(s,d)})$ and store the data in the database.

For reliable data exchange between source and destination motes, the source motes should transmit packets through the links with low packet loss. In order to choose an optimal route for each mote, the management tier collects the SNR along with the packet loss for each link. An adaptation approach should choose links with a higher SNR to minimise packet loss. However, this may increase the load at some motes in the network resulting in high energy consumption. Therefore, the link selection process should also do a fair packet distribution across the links in the network. Unfair allocation may quickly drain the batteries of some motes, which reduces the lifetime of the entire network.

B.3.4 Architecture of DeltaIoT Simulator

The DeltaIoT exemplar also offers a simulator for experimentation. The simulator enables to test and compare new adaptation solutions fast. In the simulator, the activities of the network during a specified period of wall clock time can be simulated in one run; the default period is 15 minutes. The interface to apply adaptation with the simulator and the physical IoT system are fully compatibel. This allows researchers first to experiment and test a self-adaptation solution in simulation, before testing it on the physical network. Figure B.5 shows the architecture of the DeltaIoT simulator.

$Node$, $Gateway$, $Mote$, $Link$, and $Packet$ are the basic elements of the IoT network. These elements correspond to the IoT Network Tier and the Gateway Tier in Figure B.2. $NetworkManagement$ provides the functionality for collecting and processing network data and changing the network settings. This element corresponds to the Management Tier in Figure B.2. The $SimulationClient$ offers

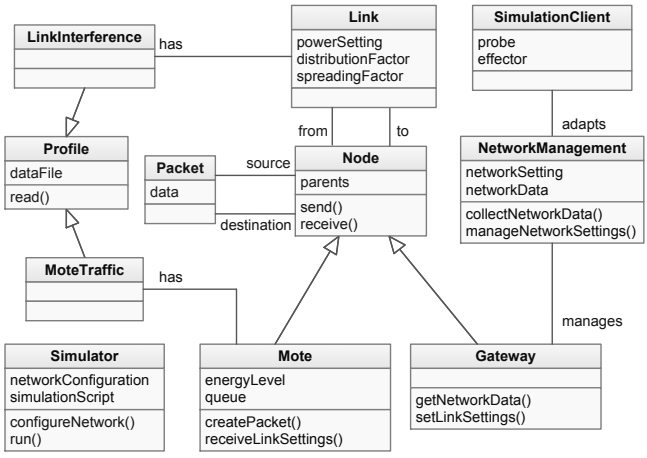


Figure B.5: Architecture of DeltaIoT Simulator

a probe and effector to apply self-adaptation to the IoT network. The probe and effector methods are identical to those in Listings B.1 and B.2.

We use the concept of a $Profile$ to specify uncertainties in the simulated system. A profile is defined by a file that contains a series of values that represent a property of the system or its environment over time. $LinkInterference$ defines the levels of interference on a link over time, while $MoteTraffic$ defines the traffic generated by a mote over time. Figure B.6 shows some example profiles of uncertainties provided by the exemplar. We collected the data for these graphs from field observations over a period of one week. The graphs at the top show fluctuations of the traffic of two motes for 24 hours; the graphs at the bottom show changes of the signal to noise ratio.

Finally, $Simulator$ enables a user to perform a simulation of a network configuration. The user can define a network configuration, i.e. topology of the IoT network with predefined network settings. The exemplar provides two predefined configurations: (i) a default network configuration with 15 motes as shown in Figure B.1, and (ii) a reference configuration where each mote in the network communicates at maximum power, and sends/forwards all its messages to all its parents. This over-provisioning approach is common in practice to assure high packet delivery performance at the cost of the lifetime of the network. A user can then define a simulation run by means of $simulationScript$. This script defines the sequence of activities of a simulation run. The exemplar provides a default script that defines a series of activities in 96 periods, each period corresponding to 15 minutes of network activity followed by an adaptation cycle.

B.4 Experimentation with DeltaIoT

Evaluating adaptive solutions is a four-step process, which we illustrate below using a simple example.

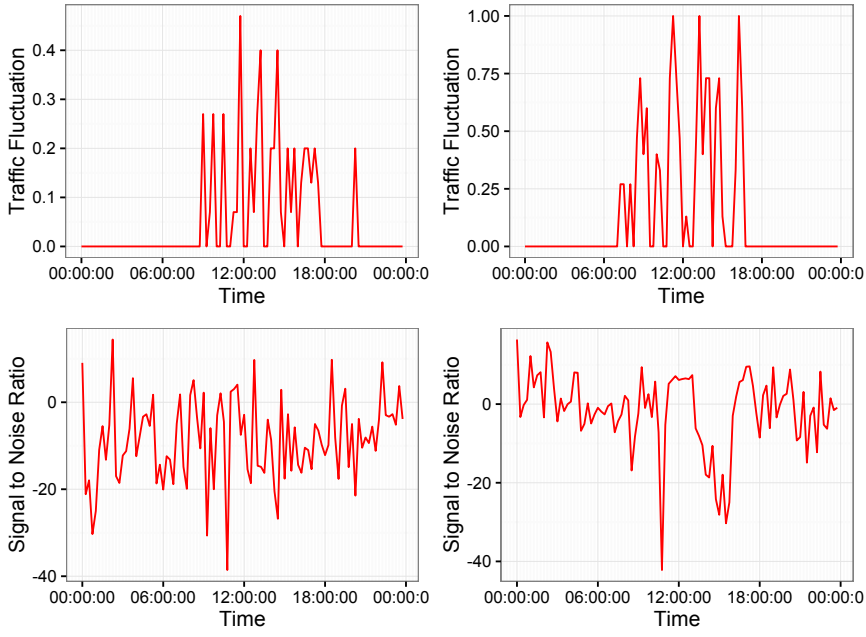


Figure B.6: Profiles of uncertainties for two motes in Figure B.1 (mote 10 left, mote 13 right)

Step 1: Download the exemplar software and supporting material at the exemplar website. Register to get the credentials to get remote access to the DeltaIoT network. Define the self-adaptation problem.

Step 2: Design a self-adaptation solution; i.e. model and implement a MAPE-K feedback loop that is connected with the probe and effector.

Step 3: Test the solution using the DeltaIoT simulator. Users can download existing solutions from the website to compare the newly developed solution.

Step 4: Apply and evaluate the solution with the physical network setup. To that end, the user has to use the credentials she/he received after registration.

Example. For step 1, we consider the adaption problem of scenario two in Table B.1: reduce packet loss and minimise energy consumption when dealing with communication interference and fluctuating traffic. We use the network topology shown in Figure B.1 and the default settings of the IoT network and fixed the spreading factor to 11.

In step 2, we design a self-adaptation solution consisting of a MAPE-K feedback loop. We briefly explain the functions of each of the MAPE elements; the Appendix added to this paper gives the pseudo code of the MAPE-K loop implementation. The Monitor periodically checks the status of the motes via the probe and updates the Knowledge accordingly. It then invokes the Analyser. The Analyzer checks for each link in the network the signal to noise ration, the power setting, and the distribution factor. If any of the settings are not optimal, the Anal-

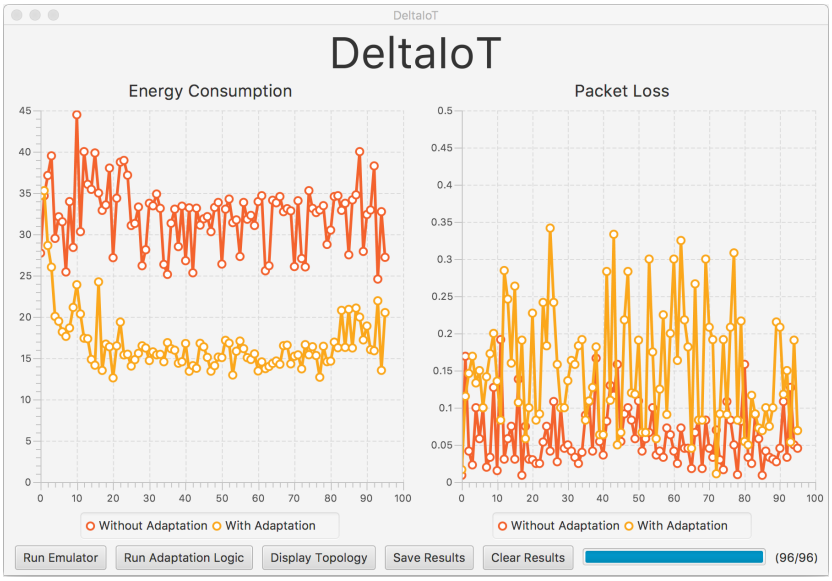


Figure B.7: Simulation results of the simple self-adaptation solution

yser invokes the Planner. The Planner generates a plan that gradually improves the network settings of the motes that are not optimal and invokes the Executor. The Executor completes the cycle by invoking the effector to apply the adaptations to the IoT network.

In step 3, we test the self-adaptation solution using the DeltaIoT simulator and compare the adaption approach with a reference approach (for the latter, each mote communicates at maximum power, and sends/forwards all its packets to all its parents). Figure B.7 shows test results for a run that corresponds with a period of one day (96 periods of 15 minutes, with self-adaptation applied after each period). The results show that the self-adaptation approach significantly reduces energy consumption compared to the reference approach. The cost however, is an increase in the packet delivery performance.

In step 4, we test the self-adaptation solution using the physical DeltaIoT setup. We compare the simple self-adaption solution with the reference approach for a run of half a day (48 periods of 15 minutes, with self-adaptation applied after each period). The test results in Figure B.8 show that the simple self-adaptation approach reduces the energy consumption (mean value of 22 compared to 33 for the reference approach) at the cost of an increase of the packet loss (mean value of 0.17 compared to 0.05 for the reference approach).

The exemplar website provides additional evaluation results obtained with ActivFORMS. In ActivFORMS, self-adaptation is realised by formally specified and verified MAPE-K models that are directly executed at runtime to adapt the IoT network (using a dedicated virtual machine) [109]. ActivFORMS exploits statistical model checking at runtime to select configurations that comply with the self-adaptation goals [209].

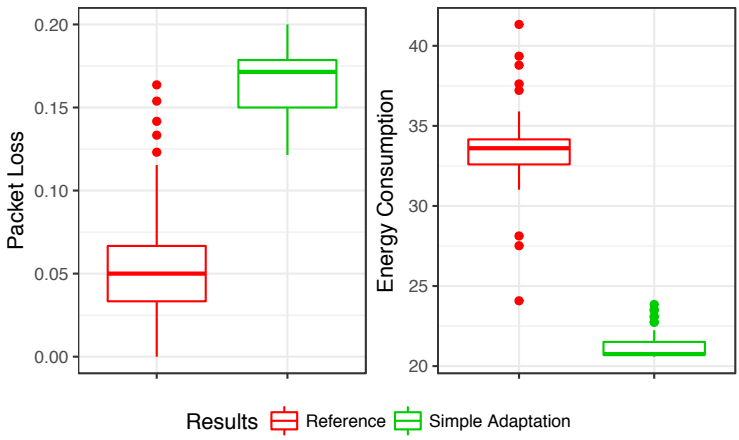


Figure B.8: Test results of the physical IoT system

B.5 Conclusions

With a growing number of IoT projects, such as smart homes, industrial facility tracking and control, and sea pollution monitoring, these system are being deployed in highly uncertain and rapidly changing environments. These uncertainties are often not completely known at deployment time, making IoT an important emerging domain for research on self-adaptation. In this paper, we presented the DeltaIoT exemplar that promotes research through enabling the comparison of different self-adaptation solutions in the domain of IoT. DeltaIoT supports researchers by reducing the time required to build, evaluate, and compare self-adaptation solutions. DeltaIoT is the first exemplar for research in self-adaptation that combines a simulator with a real world setup for experimentation. We hope that the research community will use the DeltaIoT exemplar to evaluate and compare novel solutions in adaptive and self-managing IoT systems, and drive their further development. The exemplar is available via the project website and the DARTS exemplar website: `http://dx.doi.org/10.4230/DARTS.3.1.4`.

Appendix

Listing B.3: DeltaIoT effecting methods.

```
module Monitor
 K.motes = probe.getAllmotes()
 invoke Analyzer()

module Analyzer
 foreach(mote in K.motes) {
  foreach(link in mote.links) {
    // Check whether SNR > 0 and Power == 0
    // OR SNR < 0 && Power == 15
    if (!isPowerOptimal(link))
        adaptationRequired = true;
  }
  if (mote.links > 1) {
   // check power settings of links
   if (!allLinkUseSamePower(mote))
     adaptationRequired = true;
  }
 }
 if (adaptationRequired) invoke Planner

module Planner
 foreach(mote in K.motes) {
  foreach(link in mote.Link) {
    if (!isPowerOptimal(link)) {
       if (link.SNR < 0)
        addStep(link, CHANGE_POWER, link.power+1)
       if (link.SNR > 0)
        addStep(link, CHANGE_POWER, link.power-1)
    }
   }
   if (mote.links > 1) {
    if (!allLinkUseSamePower(mote)) {
      link1 = mote.links[0]
      link2 = mote.links[1]
      if (link1.power > link2.power) {
        addStep(link, CHANGE_DIST, link1.dist-10)
        addStep(link, CHANGE_DIST, link2.dist+10)
      }
     else {
       addStep(link, CHANGE_DIST, link1.dist+10)
       addStep(link, CHANGE_DIST, link2.dist-10)
     }
    }
   }
 }
 invoke Executor

module Executor
 foreach(step in K.planningSteps) {
   if (step.type == CHANGE_POWER) {
      effector.setLinkPower(step.link, step.value)
   }
   else if (step.type == CHANGE_DIST) {
       effector.setLinkDistribution
                         (step.link, step.value)
   }
 }
```

Appendix C

List of Publications

Following papers are part of this thesis:

- Iftikhar, M. U., & Weyns, D. (2014). ActivFORMS: Active FORmal Models for Self-adaptation. In Proceedings of the *9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'14)* (pp. 125-134). ACM. [Google scholar citations till October 2017: 58]
- Calinescu, R., Gerasimou, S., Habli, I., Iftikhar, M. U., Kelly, T., & Weyns, D. (2017). Engineering Trustworthy Self-Adaptive Software with Dynamic Assurance Cases. In *IEEE Transactions on Software Engineering (TSE)*. arXiv preprint arXiv:1703.06350. IEEE.
- Weyns, D., & Iftikhar, M. U. (2016). Model-Based Simulation at Runtime for Self-Adaptive Systems. In *IEEE International Conference on Autonomic Computing (ICAC'16)*, (pp. 364-373). IEEE.
- Weyns, D., & Iftikhar, M. U. (2017). ActivFORMS: An Efficient Approach to Engineer Self-Adaptive Systems with Guarantees. Submitted under review.
- Iftikhar, M. U., Lundberg, J., & Weyns, D. (2016). A Model Interpreter for Timed Automata. In *International Symposium on Leveraging Applications of Formal Methods (ISoLA'16)* (pp. 243-258). Springer.
- Iftikhar, M. U., Ramachandran, G. S., Bollanse, P., Weyns, D., & Hughes, D. (2017). DeltaIoT: A Self-adaptive Internet of things Exemplar. In Proceedings of the *12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'17)* (pp. 76-82). IEEE.

Additional research publications that are not included in this thesis:

- Iftikhar, M. U., & Weyns, D. (2017), ActivFORMS: A Runtime Environment for Architecture-Based Adaptation with Guarantees, In *IEEE International Conference on Software Architecture Workshops (ICSAW'17)*, pp. 278-281. IEEE.
- Algabroun, H., Iftikhar, M. U., Al-Najja, B., & Weyns, D. (2017). Maintenance 4.0 Framework Using Self-Adaptive Software Architecture. In *Second International Conference on Maintenance Engineering (IncoME-II'17)*.
- Iftikhar, M. U., & Weyns, D. (2016). Towards runtime statistical model checking for self-adaptive systems. Technical report published in *Department of Computer Science, KU Leuven, Belgium*. CW Reports vol:CW693.
- Abbas, N., Andersson, J., Iftikhar, M. U., & Weyns, D. (2016). Rigorous Architectural Reasoning for Self-Adaptive Software Systems. In *1st Workshop on Qualitative Reasoning about Software Architectures (QRASA)*, (pp. 11-18). IEEE.
- Shevtsov, S., Iftikhar, M. U., & Weyns, D. (2015). SimCA vs ActivFORMS: Comparing Control- and Architecture-Based Adaptation on the TAS Exemplar. In Proceedings of the *1st International Workshop on Control Theory for Software Engineering (CTSE15)* (pp. 1-8). ACM.
- Iftikhar, M. U., & Weyns, D. (2014). Assuring System Goals under Uncertainty with Active Formal Models of Self-Adaptation. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE'14)* (pp. 604-605). ACM.

- Weyns, D., Iftikhar, M. U., & Sderlund, J. (2013). Do External Feedback Loops Improve the Design of Self-Adaptive Systems? A Controlled Experiment. In Proceedings of the *8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'13)* (pp. 3-12). IEEE. [Google scholar citations till October 2017: 25]
- Iftikhar, M. U., & Weyns, D. (2012). A Case Study on Formal Verification of Self-Adaptive Behaviors in a Decentralized System. In Proceedings of the *11th International Workshop on Foundations of Coordination Languages and Self Adaptation (FOCLASA'12)* arXiv preprint arXiv:1208.4635. EPTCS. [Google scholar citations till October 2017: 42]
- Weyns, D., Iftikhar, M. U., De La Iglesia, D. G., & Ahmad, T. (2012). A Survey of Formal Methods in Self-Adaptive Systems. In Proceedings of the *Fifth International C* Conference on Computer Science and Software Engineering (C3S2E '12)* (pp. 67-79). ACM. [Google scholar citations till October 2017: 90]
- Weyns, D., Iftikhar, M. U., Malek, S., & Andersson, J. (2012). Claims and Supporting Evidence for Self-Adaptive Systems: A Literature Study. In Proceedings of the *7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'12)* (pp. 89-98). IEEE. [Google scholar citations till October 2017: 47]
- Iftikhar, M. U., & Weyns, D. (2012). Model Checking of Self-Adaptive Behaviors in a Multi-Agent System for Traffic Monitoring. In *10th European Workshop on Multi-Agent Systems (EUMAS'12)*.

Bibliography

[1] *ActivFORMS: Active Formal Models for Self-Adaptation*. https://people.cs.kuleuven.be/~danny.weyns/software/ActivFORMS/. 2016.

[2] I. F. Akyildiz et al. "A survey on sensor networks". In: *IEEE Communications Magazine* 40.8 (Aug. 2002), pp. 102–114. ISSN: 0163-6804. DOI: 10.1109/MCOM.2002.1024422.

[3] J. Almeida et al. "Resource Management in the Autonomic Service-Oriented Architecture". In: *2006 IEEE International Conference on Autonomic Computing*. June 2006, pp. 84–92. DOI: 10.1109/ICAC.2006.1662385.

[4] R. Alur and D. L. Dill. "A theory of timed automata". In: *Theoretical Computer Science* 126.2 (1994), pp. 183–235. ISSN: 0304-3975.

[5] M. Anlauff. "Abstract State Machines - Theory and Applications: International Workshop, ASM 2000 Monte Verità, Switzerland, March 19–24, 2000 Proceedings". In: Springer, 2000. Chap. XASM- An Extensible, Component-Based Abstract State Machines Language, pp. 69–90. ISBN: 978-3-540-44518-0. DOI: 10.1007/3-540-44518-8_6.

[6] P. Arcaini, E. Riccobene, and P. Scandurra. "Formal Design and Verification of Self-Adaptive Systems with Decentralized Control". In: *ACM Transactions on Autonomous and Adaptive System* 11.4 (Jan. 2017), 25:1–25:35. ISSN: 1556-4665. DOI: 10.1145/3019598.

[7] *Assurances for Self-Adaptive Systems - Principles, Models, and Techniques*. Vol. 7740. Lecture Notes in Computer Science. Springer, 2013. ISBN: 978-3-642-36248-4.

[8] A. Augustin et al. "A Study of LoRa: Long Range and Low Power Networks for the Internet of Things". In: *Sensors* 16(9) (2016).

[9] M. Autili, P. Inverardi, and M. Tivoli. "Automated Integration of Service-Oriented Software Systems". In: *International Conference on Fundamentals of Software Engineering, FSEN*. Springer, 2015.

[10] A. Aziz et al. "Model-checking continuous-time Markov chains". In: *ACM Transactions on Computational Logic* 1.1 (2000), pp. 162–170. ISSN: 1529-3785.

[11] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. ISBN: 026202649X, 9780262026499.

[12] C. Baier et al. "Model-Checking Algorithms for Continuous-Time Markov Chains". In: *IEEE Transactions on Software Engineering* 29.6 (2003), pp. 524–541.

[13] L. Baresi, L. Pasquale, and P. Spoletini. "Fuzzy Goals for Requirements-Driven Adaptation". In: *Proceedings of the 2010 18th IEEE International Requirements Engineering Conference*. RE '10. IEEE Computer Society, 2010, pp. 125–134. ISBN: 978-0-7695-4162-4. DOI: 10.1109/RE.2010.25.

[14] C. Barna et al. "Hogna: A Platform for Self-adaptive Applications in Cloud Environments". In: *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '15. IEEE, 2015.

[15] B. Becker et al. "Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation". In: *28th International Conference on Software Engineering*. ACM, 2006, pp. 72–81. ISBN: 1-59593-375-1. DOI: 10.1145/1134285.1134297.

[16] G. Behrmann, A. David, and K. G. Larsen. "A Tutorial on UPPAAL". In: *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*. LNCS 3185. Springer, Sept. 2004, pp. 200–236.

[17] G. Behrmann et al. "UPPAAL 4.0". In: *QEST'06*. 2006, pp. 125–126.

[18] N. Bencomo et al. "Genie: Supporting the Model Driven Development of Reflective, Component-based Adaptive Systems". In: *Proceedings of the 30th International Conference on Software Engineering*. ICSE '08. ACM, 2008. ISBN: 978-1-60558-079-1.

[19] J. Bengtsson and W. Yi. "Lectures on Concurrency and Petri Nets: Advances in Petri Nets". In: Springer, 2004. Chap. Timed Automata: Semantics, Algorithms and Tools, pp. 87–124. ISBN: 978-3-540-27755-2. DOI: 10.1007/978-3-540-27755-2_3.

[20] M. Benjamin et al. "Autonomy for Unmanned Marine Vehicles with MOOS-IvP". In: *Marine Robot Autonomy*. Springer, 2013, pp. 47–90. ISBN: 978-1-4614-5658-2. DOI: 10.1007/978-1-4614-5659-9_2.

[21] A. Bianco and L. de Alfaro. "Model checking of probabilistic and nondeterministic systems". In: *FSTTCS'95*. 1995, pp. 499–513.

[22] P. Bishop and R. Bloomfield. "A Methodology for Safety Case Development". In: *Industrial Perspectives of Safety-critical Systems*. Springer, 1998, pp. 194–203. ISBN: 978-3-540-76189-1. DOI: 10.1007/978-1-4471-1534-2_14.

[23] G. Blair, N. Bencomo, and R. B. France. "Models@ run.time". In: *Computer* 42.10 (Oct. 2009), pp. 22–27. ISSN: 0018-9162. DOI: 10.1109/MC.2009.326.

[24] R. Bloomfield and P. Bishop. “Safety and Assurance Cases: Past, Present and Possible Future — an Adelard Perspective”. In: *Making Systems Safer*. Springer, 2010, pp. 51–67. ISBN: 978-1-84996-085-4. DOI: 10.1007/978-1-84996-086-1_4.

[25] P. Bordia et al. “Uncertainty during organizational change: Types, consequences, and management strategies”. In: *Journal of business and psychology* 18.4 (2004), pp. 507–532.

[26] V. Braberman et al. “Controller Synthesis: From Modelling to Enactment”. In: *35th International Conference on Software Engineering*. 2013, pp. 1347–1350. ISBN: 978-1-4673-3076-3.

[27] O. Brukman, S. Dolev, and E. K. Kolodner. “A Self-stabilizing Autonomic Recoverer for Eventual Byzantine Software”. In: *Journal of Systems and Software* 81.12 (Dec. 2008), pp. 2315–2327. ISSN: 0164-1212. DOI: 10.1016/j.jss.2008.04.028.

[28] Y. Brun et al. “Engineering Self-Adaptive Systems Through Feedback Loops”. In: *Software Engineering for Self-Adaptive Systems*. Springer, 2009, pp. 48–70. ISBN: 978-3-642-02160-2. DOI: 10.1007/978-3-642-02161-9_3.

[29] R. Buizza, M. Milleer, and T. Palmer. “Stochastic representation of model uncertainties in the ECMWF ensemble prediction system”. In: *Quarterly Journal of the Royal Meteorological Society* 125.560 (1999), pp. 2887–2908.

[30] A. Cailliau and A. V. Lamsweerde. “Runtime Monitoring and Resolution of Probabilistic Obstacles to System Goals”. In: *12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 2017.

[31] R. Calinescu, K. Johnson, and Y. Rafiq. “Developing self-verifying service-based systems”. In: *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. Nov. 2013, pp. 734–737. DOI: 10.1109/ASE.2013.6693145.

[32] R. Calinescu, K. Johnson, and Y. Rafiq. “Using Observation Ageing to Improve Markovian Model Learning in QoS Engineering”. In: *2nd ACM/SPEC International Conference on Performance Engineering*. 2011, pp. 505–510.

[33] R. Calinescu. “General-Purpose Autonomic Computing”. In: *Autonomic Computing and Networking*. Springer, 2009, pp. 3–30.

[34] R. Calinescu, S. Gerasimou, and A. Banks. “Self-adaptive Software with Decentralised Control Loops”. In: *FASE’15*. Vol. 9033. LNCS. Springer, 2015, pp. 235–251. ISBN: 978-3-662-46674-2. DOI: 10.1007/978-3-662-46675-9_16.

[35] R. Calinescu and M. Z. Kwiatkowska. “Using Quantitative Analysis to Implement Autonomic IT Systems”. In: *ICSE’09*. 2009, pp. 100–110.

[36] R. Calinescu et al. “Adaptive Model Learning for Continual Verification of Non-functional Properties”. In: *5th ACM/SPEC International Conference on Performance Engineering*. 2014, pp. 87–98.

[37] R. Calinescu et al. “Dynamic QoS Management and Optimization in Service-Based Systems”. In: *IEEE Transactions on Software Engineering* 37.3 (May 2011), pp. 387–409. ISSN: 0098-5589. DOI: 10.1109/TSE.2010.92.

[38] R. Calinescu et al. “Self-adaptive software needs quantitative verification at runtime”. In: *Communications of the ACM* 55.9 (Sept. 2012), pp. 69–77. ISSN: 0001-0782. DOI: 10.1145/2330667.2330686.

[39] R. Calinescu et al. “Synthesis and Verification of Self-aware Computing Systems”. In: *Self-Aware Computing Systems*. Springer, 2017, pp. 337–373. ISBN: 978-3-319-47474-8. DOI: 10.1007/978-3-319-47474-8_11.

[40] R. Calinescu et al. “Engineering Trustworthy Self-Adaptive Software with Dynamic Assurance Cases”. In: *IEEE Transactions on Software Engineering* (© 2017 IEEE. Reprinted, with permission).

[41] J. Cámara et al. “Analyzing Latency-Aware Self-Adaptation Using Stochastic Games and Simulations”. In: *ACM Transactions on Autonomous and Adaptive Systems* 10.4 (Jan. 2016), 23:1–23:28. ISSN: 1556-4665. DOI: 10.1145/2774222.

[42] J. Cámara et al. “Optimal planning for architecture-based self-adaptation via model checking of stochastic games”. In: *30th Annual ACM Symposium on Applied Computing*. 2015, pp. 428–435.

[43] J. Camara et al. “Robustness-Driven Resilience Evaluation of Self-Adaptive Software Systems”. In: *IEEE Transactions on Dependable and Secure Computing* 14.1 (2017), pp. 50–64.

[44] M. Caporuscio et al. “GoPrime: A Fully Decentralized Middleware for Utility-Aware Service Assembly”. In: *IEEE Transactions on Software Engineering* 42.2 (Feb. 2016), pp. 136–152. ISSN: 0098-5589. DOI: 10.1109/TSE.2015.2476797.

[45] A. Cerpa and D. Estrin. “ASCENT: adaptive self-configuring sensor networks topologies”. In: *IEEE Transactions on Mobile Computing* 3.3 (July 2004), pp. 272–285. ISSN: 1536-1233. DOI: 10.1109/TMC.2004.16.

[46] T. Chen et al. “Automatic Verification of Competitive Stochastic Systems”. In: *Proc. 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’12)*. Vol. 7214. LNCS. Springer, 2012, pp. 315–330.

[47] B. H. C. Cheng and J. M. Atlee. “Research Directions in Requirements Engineering”. In: *Future of Software Engineering*. IEEE, 2007, pp. 285–303. ISBN: 0-7695-2829-5.

[48] B. H. Cheng et al. “Software Engineering for Self-Adaptive Systems”. In: Springer, 2009. Chap. Software Engineering for Self-Adaptive Systems: A Research Roadmap, pp. 1–26. ISBN: 978-3-642-02160-2. DOI: 10.1007/978-3-642-02161-9_1.

[49] B. H. C. Cheng et al. “Models@run.time: Foundations, Applications, and Roadmaps”. In: Springer, 2014. Chap. Using Models at Runtime to Address Assurance for Self-Adaptive Systems, pp. 101–136. ISBN: 978-3-319-08915-7. DOI: 10.1007/978-3-319-08915-7_4.

[50] S. W. Cheng, D. Garlan, and B. Schmerl. “Evaluating the effectiveness of the Rainbow self-adaptive system”. In: *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*. May 2009, pp. 132–141. DOI: 10.1109/SEAMS.2009.5069082.

[51] E. M. Clarke et al. “Statistical Model Checking in BioLab: Applications to the Automated Analysis of T-Cell Receptor Signaling Pathway”. In: *6th International Conference on Computational Methods in Systems Biology*. Springer, 2008.

[52] E. M. Clarke, E. A. Emerson, and A. P. Sistla. “Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications”. In: *ACM Transactions on Programming Languages and Systems* 8.2 (Apr. 1986), pp. 244–263. ISSN: 0164-0925. DOI: 10.1145/5397.5399.

[53] E. M. Clarke et al. “Model Checking and the State Explosion Problem”. In: *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*. Springer, 2012, pp. 1–30. ISBN: 978-3-642-35746-6. DOI: 10.1007/978-3-642-35746-6_1.

[54] E. Clarke, D. Long, and K. McMillan. “Compositional model checking”. In: *Proceedings. Fourth Annual Symposium on Logic in Computer Science*. 1989, pp. 353–362.

[55] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. “Learning assumptions for compositional verification”. In: *9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2003, pp. 331–346.

[56] Common Criteria Recognition Arrangement. *ISO/IEC 15408 – Common Criteria for Information Technology Security Evaluation, Version 3.1, Revision 4*. Sept. 2012.

[57] A. Computing et al. “An architectural blueprint for autonomic computing”. In: *IBM White Paper* 31 (2006).

[58] M. Cordy et al. “Model Checking Adaptive Software with Featured Transition Systems”. In: *Assurances for Self-Adaptive Systems*. Vol. 7740. LNCS. Springer, 2013, pp. 1–29. ISBN: 978-3-642-36248-4. DOI: 10.1007/978-3-642-36249-1_1.

[59] A. David et al. “Uppaal SMC Tutorial”. In: *International Journal on Software Tools for Technology Transfer* 17.4 (Aug. 2015), pp. 397–415. ISSN: 1433-2779. DOI: 10.1007/s10009-014-0361-y.

[60] E. Denney and G. Pai. “A Formal Basis for Safety Case Patterns”. In: *Comp. Safety, Reliability, and Security*. Vol. 8153. LNCS. Springer, 2013, pp. 21–32. ISBN: 978-3-642-40792-5. DOI: 10.1007/978-3-642-40793-2_3.

[61] E. Denney, I. Habli, and G. Pai. “Dynamic Safety Cases for Through-life Safety Assurance”. In: *ICSE’15*. 2015, pp. 587–590.

[62] N. R. D’Ippolito et al. “Synthesis of Live Behaviour Models”. In: *18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2010, pp. 77–86. ISBN: 978-1-60558-791-2. DOI: 10.1145/1882291.1882305.

[63] N. D’Ippolito et al. “Hope for the Best, Prepare for the Worst: Multi-tier Control for Adaptive Systems”. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. ACM, 2014, pp. 688–699. ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568264.

[64] N. D’Ippolito et al. “Synthesis of live behaviour models for fallible domains”. In: *33rd International Conference on Software Engineering*. 2011, pp. 211–220. DOI: 10.1145/1985793.1985823.

[65] S. Dobson et al. “A Survey of Autonomic Communications”. In: *ACM Transactions on Autonomous and Adaptive System* 1.2 (Dec. 2006), pp. 223–259. ISSN: 1556-4665.

[66] D. Dujovne et al. “6TiSCH: deterministic IP-enabled industrial internet (of things)”. In: *IEEE Communications Magazine* 52.12 (Dec. 2014), pp. 36–41. ISSN: 0163-6804. DOI: 10.1109/MCOM.2014.6979984.

[67] A. Elkhodary, N. Esfahani, and S. Malek. “FUSION: a framework for engineering self-tuning self-adaptive software systems”. In: *FSE’10*. 2010, pp. 7–16.

[68] I. Epifani et al. “Model evolution by Run-time Parameter Adaptation”. In: *31st International Conference on Software Engineering*. 2009, pp. 111–121. DOI: 10.1109/ICSE.2009.5070513.

[69] N. Esfahani, E. Kouroshfar, and S. Malek. “Taming uncertainty in self-adaptive software”. In: *ESEC/FSE*. 2011. ISBN: 978-1-4503-0443-6. DOI: 10.1145/2025113.2025147.

[70] European Organisation for the Safety of Air Navigation. *Safety Case Development Manual*. 2006.

[71] A. Filieri, C. Ghezzi, and G. Tamburrelli. “A formal approach to adaptive software: continuous assurance of non-functional requirements”. In: *Formal Asp. Comput.* 24.2 (2012), pp. 163–186.

[72] A. Filieri, C. Ghezzi, and G. Tamburrelli. “Run-time Efficient Probabilistic Model Checking”. In: *ICSE’11*. 2011, pp. 341–350. DOI: 10.1145/1985793.1985840.

[73] A. Filieri, G. Tamburrelli, and C. Ghezzi. “Supporting Self-Adaptation via Quantitative Verification and Sensitivity Analysis at Run Time”. In: *IEEE Transactions on Software Engineering* 42.1 (2016), pp. 75–99.

[74] A. Filieri, L. Grunske, and A. Leva. “Lightweight Adaptive Filtering for Efficient Learning and Updating of Probabilistic Models”. In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. ICSE ’15. IEEE, 2015, pp. 200–211. ISBN: 978-1-4799-1934-5.

[75] A. Filieri, H. Hoffmann, and M. Maggio. “Automated Design of Self-adaptive Software with Control-theoretical Formal Guarantees”. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. ACM, 2014, pp. 299–310. ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568272.

[76] V. Forejt et al. “Incremental Runtime Verification of Probabilistic Systems”. In: *Runtime Verification*. Vol. 7687. LNCS. Springer, 2012, pp. 314–319.

[77] M. Fowler. *Domain-specific Languages*. Pearson Education, 2010.

[78] E. M. Fredericks, B. DeVries, and B. H. C. Cheng. “Towards Run-time Adaptation of Test Cases for Self-adaptive Systems in the Face of Uncertainty”. In: *9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. 2014, pp. 17–26. ISBN: 978-1-4503-2864-7. DOI: 10.1145/2593929.2593937.

[79] D. G. de la Iglesia and D. Weyns. “MAPE-K Formal Templates to Rigorously Design Behaviors for Self-Adaptive Systems”. In: *ACM Transactions on Autonomous and Adaptive Systems* 10.3 (Sept. 2015), 15:1–15:31. ISSN: 1556-4665. DOI: 10.1145/2724719.

[80] S. Gallotti et al. “Quality Prediction of Service Compositions through Probabilistic Model Checking”. In: *Proc. 4th International Conference on the Quality of Software-Architectures, QoSA 2008*. Vol. 5281. LNCS. Springer, 2008, pp. 119–134. ISBN: 978-3-540-87878-0.

[81] M. Galster and D. Weyns. “Empirical Research in Software Architecture: How Far have We Come?” In: *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. Apr. 2016, pp. 11–20. DOI: 10.1109/WICSA.2016.10.

[82] J. García-Galán et al. “User-centric Adaptation of Multi-tenant Services: Preference-based Analysis for Service Reconfiguration”. In: *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS 2014. ACM, 2014, pp. 65–74. ISBN: 978-1-4503-2864-7. DOI: 10.1145/2593929.2593930.

[83] D. Garlan and B. Schmerl. “Model-based adaptation for self-healing systems”. In: WOSS. 2002.

[84] D. Garlan et al. “Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure”. In: *Computer* 37.10 (Oct. 2004), pp. 46–54. ISSN: 0018-9162. DOI: 10.1109/MC.2004.175.

[85] E. Gat. “Artificial Intelligence and Mobile Robots”. In: MIT Press, 1998. Chap. Three-layer Architectures, pp. 195–210. ISBN: 0-262-61137-6.

[86] I. Georgiadis, J. Magee, and J. Kramer. “Self-organising Software Architectures for Distributed Systems”. In: *Proceedings of the First Workshop on Self-healing Systems*. WOSS ’02. ACM, 2002, pp. 33–38. ISBN: 1-58113-609-9. DOI: 10.1145/582128.582135.

[87] S. Gerasimou, R. Calinescu, and A. Banks. “Efficient Runtime Quantitative Verification Using Caching, Lookahead, and Nearly-optimal Reconfiguration”. In: *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS 2014. ACM, 2014, pp. 115–124. ISBN: 978-1-4503-2864-7. DOI: 10.1145/2593929.2593932.

[88] S. Gerasimou, G. Tamburrelli, and R. Calinescu. “Search-Based Synthesis of Probabilistic Models for Quality-of-Service Software Engineering”. In: *30th International Conference Automated Software Engineering (ASE’15)*. 2015.

[89] S. Gerasimou et al. “UNDERSEA: An Exemplar for Engineering Self-Adaptive Unmanned Underwater Vehicles (Artifact)”. In: *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 2017.

[90] L. Gherardi and N. Hochgeschwender. “RRA: Models and tools for robotics run-time adaptation”. In: *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*. Sept. 2015, pp. 1777–1784. DOI: 10.1109/IROS.2015.7353608.

[91] C. Ghezzi et al. “Managing Non-functional Uncertainty via Model-driven Adaptivity”. In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE ’13. IEEE, 2013, pp. 33–42. ISBN: 978-1-4673-3076-3.

[92] C. Ghezzi et al. “Mining Behavior Models from User-intensive Web Applications”. In: *36th International Conference on Software Engineering*. 2014, pp. 277–287.

[93] D. Gil de la Iglesia. “A formal approach for designing distributed self-adaptive systems”. PhD thesis. Linnaeus University Press, 2014.

[94] D. Gil de la Iglesia and D. Weyns. “Guaranteeing robustness in a mobile learning application using formally verified MAPE loops”. In: *2013 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. SEAMS. 2013. ISBN: 978-1-4673-4401-2.

[95] GSN Working Group Online. *Goal Structuring Notation Standard, Version 1*. Nov. 2011.

[96] H. Hansson and B. Jonsson. “A Logic for Reasoning about Time and Reliability”. In: *Formal Aspects of Computing* 6.5 (1994), pp. 512–535.

[97] K. Havelund et al. “Formal Methods for Real-Time and Probabilistic Systems”. In: Springer. Chap. Formal Verification of a Power Controller Using the Real-Time Model Checker Uppaal, pp. 277–298. ISBN: 978-3-540-48778-4. DOI: 10.1007/3-540-48778-6_17.

[98] R. Hawkins, I. Habli, and T. Kelly. “The principles of software safety assurance”. In: *31st International System Safety Conference*. 2013.

[99] R. Hawkins, I. Habli, and T. Kelly. “Principled Construction of Software Safety Cases”. In: *SAFECOMP 2013 Workshop on Next Generation of System Assurance Approaches for Safety-Critical Systems*. 2013.

[100] R. Hawkins et al. “Assurance cases and prescriptive software safety certification: A comparative study”. In: *Safety Science* 59 (2013), pp. 55–71. ISSN: 0925-7535. DOI: http://dx.doi.org/10.1016/j.ssci.2013.04.007.

[101] R. Hawkins et al. “Using a software safety argument pattern catalogue: Two case studies”. In: *Comp. Safety, Reliability, and Security*. Springer, 2011, pp. 185–198.

[102] C. M. Hayden et al. “Specifying and Verifying the Correctness of Dynamic Software Updates”. In: *International Conference on Verified Software: Theories, Tools, Experiments*. VSTTE’12. Springer, 2012, pp. 278–293. ISBN: 978-3-642-27704-7. DOI: 10.1007/978-3-642-27705-4_22.

[103] A. Hessel et al. “Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers”. In: Springer, 2008. Chap. Testing Real-Time Systems Using UPPAAL, pp. 77–117. ISBN: 978-3-540-78917-8. DOI: 10.1007/978-3-540-78917-8_3.

[104] C. M. Holloway. “Why Engineers Should Consider Formal Methods”. In: *In 1997 AIAA/IEEE 16th Digital Avionics Systems Conference*. 1997, p. 9.

[105] P. Horn. *Autonomic computing: IBM’s Perspective on the State of Information Technology*. 2001.

[106] M. C. Huebscher and J. A. McCann. “A survey of autonomic computing—degrees, models, and applications”. In: *ACM Computer Survey* 40.3 (2008), pp. 1–28. ISSN: 0360-0300.

[107] M. U. Iftikhar, J. Lundberg, and D. Weyns. “A Model Interpreter for Timed Automata”. In: *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques: 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10–14, 2016, Proceedings, Part I*. Springer, 2016, pp. 243–258. ISBN: 978-3-319-47166-2, 2016. With permission of Springer.

[108] M. U. Iftikhar and D. Weyns. “A Case Study on Formal Verification of Self-Adaptive Behaviors in a Decentralized System”. In: *Foundations of Coordination Languages and Self Adaptation, FOCLASA, ArXiv e-prints* (Aug. 2012). arXiv: 1208.4635 [cs.SE].

[109] M. U. Iftikhar and D. Weyns. “ActivFORMS: Active FORmal Models for Self-adaptation”. In: *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS 2014. ACM, 2014, pp. 125–134. ISBN: 978-1-4503-2864-7, © 2014 ACM, Inc. Reprinted by permissions, https://doi.org/10.1145/2593929.2593944.

[110] M. U. Iftikhar and D. Weyns. “Towards runtime statistical model checking for self-adaptive systems”. In: (2016).

[111] M. U. Iftikhar et al. “DeltaIoT: A Self-Adaptive Internet of Things Exemplar”. In: *12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. © 2017 IEEE. Reprinted, with permission.

[112] J. Zhang and B. Cheng. “Model-based development of dynamically adaptive software”. In: *28th International Conference on Software Engineering*. ACM, 2006.

[113] K. Johnson, R. Calinescu, and S. Kikuchi. “An Incremental Verification Framework for Component-Based Software Systems”. In: *16th International ACM Sigsoft Symposium on Component-Based Software Engineering*. 2013, pp. 33–42.

[114] G. Karsai and J. Sztipanovits. “A model-based approach to self-adaptive software”. In: *Intelligent Systems and their Applications, IEEE* 14.3 (May 1999), pp. 46–53. ISSN: 1094-7167. DOI: 10.1109/5254.769884.

[115] T. Kelly and R. Weaver. “The Goal Structuring Notation – A Safety Argument Notation”. In: *Assurance Cases Workshop*. 2004.

[116] J. Kephart. “Research challenges of autonomic computing”. In: *International Conference on Software Engineering*. 2005.

[117] J. Kephart and D. Chess. “The Vision of Autonomic Computing”. In: *Computer* 36.1 (Jan. 2003), pp. 41–50. ISSN: 0018-9162. DOI: 10.1109/MC.2003.1160055.

[118] N. Khakpour, F. Arbab, and E. Rutten. “Synthesizing structural and behavioral control for reconfigurations in component-based systems”. In: *Formal Aspects of Computing* 28.1 (2016), pp. 21–43. ISSN: 1433-299X. DOI: 10.1007/s00165-015-0346-y.

[119] R. Khan et al. “Future Internet: The Internet of Things Architecture, Possible Applications and Key Challenges”. In: *2012 10th International Conference on Frontiers of Information Technology*. Dec. 2012, pp. 257–260. DOI: 10.1109/FIT.2012.53.

[120] M. Kit et al. “An Architecture Framework for Experimentations with Self-adaptive Cyber-physical Systems”. In: *Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS ’15. IEEE, 2015.

[121] J. Knight, J. Rowanhill, and J. Xiang. “A Safety Condition Monitoring System”. In: *3rd International Workshop on Assurance Cases for Software Intensive Systems*. 2015.

[122] J. Kramer and J. Magee. “Self-Managed Systems: an Architectural Challenge”. In: *Future of Software Engineering, 2007. FOSE ’07*. May 2007, pp. 259–268. DOI: 10.1109/FOSE.2007.19.

[123] J. Kramer and J. Magee. “The Evolving Philosophers Problem: Dynamic Change Management”. In: *IEEE Transactions on Software Engineering* 16.11 (Nov. 1990). ISSN: 0098-5589.

[124] C. M. Krishna. “Real-Time Systems”. In: *Wiley Encyclopedia of Electrical and Electronics Engineering*. John Wiley & Sons, Inc., 2001. ISBN: 9780471346081. DOI: 10.1002/047134608X.W1683.

[125] C. Krupitzer et al. “A survey on engineering approaches for self-adaptive systems”. In: *Pervasive and Mobile Computing* 17, Part B (2015), pp. 184–206. ISSN: 1574-1192. DOI: http://dx.doi.org/10.1016/j.pmcj.2014.09.009.

[126] M. Kwiatkowska, G. Norman, and D. Parker. “PRISM 4.0: Verification of Probabilistic Real-time Systems”. In: *Computer aided verification*. Springer. 2011, pp. 585–591.

[127] M. Lahijanian, S. B. Andersson, and C. Belta. “Formal Verification and Synthesis for Discrete-Time Stochastic Systems”. In: *IEEE Transactions on Automatic Control* 60.8 (2015), pp. 2031–2045. DOI: 10.1109/TAC.2015.2398883.

[128] Y. Laurent et al. “Formalization of fUML: An Application to Process Verification”. In: *Advanced Information Systems Engineering*. Springer. 2014, pp. 347–363.

[129] A. Legay and B. Delahaye. “Statistical Model Checking : An Overview”. In: *CoRR* abs/1005.1327 (2010).

[130] A. Legay, B. Delahaye, and S. Bensalem. “Statistical Model Checking: An Overview.” In: *RV* 10 (2010), pp. 122–135.

[131] A. Legay, S. Sedwards, and L.-M. Traonouez. “Rare Events for Statistical Model Checking an Overview”. In: *Reachability Problems*. Springer, 2016.

[132] R. de Lemos et al. “Software Engineering for Self-adaptive Systems: Research Challenges in the Provision of Assurances”. In: *Software Engineering for Self-Adaptive Systems III*. Lecture Notes in Computer Science vol. 9640: Springer, 2017.

[133] R. de Lemos et al. “Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers”. In: Springer, 2013. Chap. Software Engineering for Self-Adaptive Systems: A Second Research Roadmap, pp. 1–32. ISBN: 978-3-642-35813-5. DOI: 10.1007/978-3-642-35813-5_1.

[134] R. de Lemos et al. “Software Engineering for Self-Adaptive Systems: A Second Research Roadmap”. In: *Software Engineering for Self-Adaptive Systems II*. Vol. 7475. LNCS. Springer, 2013, pp. 1–32. ISBN: 978-3-642-35812-8. DOI: 10.1007/978-3-642-35813-5_1.

[135] R. de Lemos et al. “Software Engineering for Self-Adaptive Systems: Assurances (Dagstuhl Seminar 13511)”. In: *Dagstuhl Reports* 3.12 (2014), pp. 67–96. ISSN: 2192-5283. DOI: http://dx.doi.org/10.4230/DagRep.3.12.67.

[136] L. Li et al. “Modeling and Analyzing the Reliability and Cost of Service Composition in the IoT: A Probabilistic Approach”. In: *2012 IEEE 19th International Conference on Web Services*. June 2012, pp. 584–591. DOI: 10.1109/ICWS.2012.25.

[137] B. Littlewood and D. Wright. “The Use of Multilegged Arguments to Increase Confidence in Safety Claims for Software-Based Systems”. In: *IEEE Transactions on Software Engineering* 33.5 (2007), pp. 347–365.

[138] C. Lu et al. “Feedback Control Real-Time Scheduling: Framework, Modeling, and Algorithms*”. In: *Real-Time Systems* 23.1 (2002), pp. 85–126. ISSN: 1573-1383. DOI: 10.1023/A:1015398403337.

[139] F. D. Macías-Escrivá et al. “Self-adaptive systems: A survey of current approaches, research challenges and applications”. In: *Expert Systems with Applications* 40.18 (2013), pp. 7267–7279. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2013.07.033.

[140] J. Magee and T. Maibaum. “Towards Specification, Modelling and Analysis of Fault Tolerance in Self Managed Systems”. In: *Proceedings of the 2006 International Workshop on Self-adaptation and Self-managing Systems*. SEAMS ’06. ACM, 2006, pp. 30–36. ISBN: 1-59593-403-0. DOI: 10.1145/1137677.1137684.

[141] S. Mahdavi-Hezavehi et al. “A systematic literature review on methods that handle multiple quality attributes in architecture-based self-adaptive systems”. In: *Information and Software Technology* 90.Supplement C (2017), pp. 1–26. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2017.03.013.

[142] V. P. L. Manna et al. “Formalizing correctness criteria of dynamic updates derived from specification changes”. In: *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. 2013, pp. 63–72.

[143] S. J. Mellor and M. Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0201748045.

[144] N. Metropolis et al. “Equation of state calculations by fast computing machines”. In: *The journal of chemical physics* 21.6 (1953), pp. 1087–1092.

[145] G. A. Moreno et al. “Proactive Self-adaptation Under Uncertainty: A Probabilistic Model Checking Approach”. In: *Foundations of Software Engineering*. ACM, 2015.

[146] J. M. Murphy et al. “Quantification of modelling uncertainties in a large ensemble of climate change simulations”. In: *Nature* 430.7001 (2004), pp. 768–772.

[147] L. Nahabedian et al. “Assured and Correct Dynamic Update of Controllers”. In: *11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2016, pp. 96–107. ISBN: 978-1-4503-4187-5.

[148] E. Y. Nakagawa et al. “The State of the Art and Future Perspectives in Systems of Systems Software Architectures”. In: *Proceedings of the First International Workshop on Software Engineering for Systems-of-Systems*. SESoS ’13. ACM, 2013, pp. 13–20. ISBN: 978-1-4503-2048-1. DOI: 10.1145/2489850.2489853.

[149] North European Functional Airspace Block. *NEFAB Project—Safety Case Report, Version 3.01*. Dec. 2011.

[150] P. Oreizy, N. Medvidovic, and R. N. Taylor. “Architecture-based Runtime Software Evolution”. In: *Proceedings of the 20th International Conference on Software Engineering*. ICSE ’98. IEEE Computer Society, 1998, pp. 177–186. ISBN: 0-8186-8368-6.

[151] P. Oreizy et al. “An Architecture-Based Approach to Self-Adaptive Software”. In: *IEEE Intelligent Systems* 14.3 (May 1999), pp. 54–62. ISSN: 1541-1672. DOI: 10.1109/5254.769885.

[152] T. J. Parr and R. W. Quong. “ANTLR: A Predicated-LL(K) Parser Generator”. In: *Software Practice and Experience* 25.7 (July 1995), pp. 789–810. ISSN: 0038-0644. DOI: 10.1002/spe.4380250705.

[153] D. Perez-Palacin, R. Calinescu, and J. Merseguer. “Log2Cloud: Log-based Prediction of Cost-performance Trade-offs for Cloud Deployments”. In: *28th Annual ACM Symposium on Applied Computing*. 2013, pp. 397–404.

[154] D. Perez-Palacin and R. Mirandola. “Uncertainties in the Modeling of Self-adaptive Systems: A Taxonomy and an Example of Availability Evaluation”. In: *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*. ICPE ’14. ACM, 2014, pp. 3–14. ISBN: 978-1-4503-2733-6. DOI: 10.1145/2568088.2568095.

[155] A. Pnueli. “The temporal logic of programs”. In: *18th Annual Symposium on Foundations of Computer Science*. 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[156] H. B. Pötter and A. Sztajnberg. “Adapting Heterogeneous Devices into an IoT Context-aware Infrastructure”. In: *Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS ’16. ACM, 2016.

[157] H. Psaier and S. Dustdar. “A survey on self-healing systems: approaches and systems”. In: *Computing* 91.1 (2011), pp. 43–73. ISSN: 0010-485X. DOI: 10.1007/s00607-010-0107-y.

[158] T. Quatmann et al. “Parameter Synthesis for Markov Models: Faster Than Ever”. In: *14th International Symposium on Automated Technology for Verification and Analysis (ATVA)*. 2016, pp. 50–67.

[159] G. Ramachandran et al. “uPnP-WAN: Application of LoRa and its deployment in DR Congo”. In: *Proceedings of the 9th International Conference on Communication Systems and Networks*. 2017.

[160] P. J. G. Ramadge and W. M. Wonham. “The control of discrete event systems”. In: *Proceedings of the IEEE* 77.1 (Jan. 1989), pp. 81–98. ISSN: 0018-9219. DOI: 10.1109/5.21072.

[161] E. Reisner et al. “Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems”. In: *ACM/IEEE International Conference on Software Engineering*. ICSE ’10. ACM, 2010, pp. 445–454. ISBN: 978-1-60558-719-6. DOI: 10.1145/1806799.1806864.

[162] Royal Academy of Engineering. *Establishing High-Level Evidence for the Safety and Efficacy of Medical Devices and Systems*. Jan. 2013.

[163] J. Rushby. *The Interpretation and Evaluation of Assurance Cases*. Tech. rep. SRI-CSL-15-01. Comp. Science Laboratory, SRI International, 2015.

[164] M. Salehie and L. Tahvildari. “Self-adaptive Software: Landscape and Research Challenges”. In: *ACM Transactions on Autonomous and Adaptive Systems* 4.2 (May 2009), 14:1–14:42. ISSN: 1556-4665. DOI: 10.1145/1516533.1516538.

[165] P. Sawyer et al. “Requirements-Aware Systems: A Research Agenda for RE for Self-adaptive Systems”. In: *2010 18th IEEE International Requirements Engineering Conference*. Sept. 2010, pp. 95–103.

[166] B. Schmerl et al. “Challenges in Composing and Decomposing Assurances for Self-Adaptive Systems”. In: *Software Engineering for Self-Adaptive Systems (SEfSAS) 3*. 9640. Springer, 2017.

[167] D. C. Schmidt. “Model-driven Engineering”. In: *COMPUTER-IEEE COMPUTER SOCIETY-* 39.2 (2006), p. 25.

[168] D. Schneider and M. Trapp. “Conditional Safety Certification of Open Adaptive Systems”. In: *ACM Transactions on Autonomous and Adaptive Systems* 8.2 (July 2013), 8:1–8:20. ISSN: 1556-4665. DOI: 10.1145/2491465.2491467.

[169] K. Shah and M. Kumar. “Distributed Independent Reinforcement Learning (DIRL) Approach to Resource Management in Wireless Sensor Networks”. In: *2007 IEEE International Conference on Mobile Adhoc and Sensor Systems*. Oct. 2007, pp. 1–9. DOI: 10.1109/MOBHOC.2007.4428658.

[170] S. Shevtsov, M. U. Iftikhar, and D. Weyns. “SimCA vs ActivFORMS: Comparing Control- and Architecture-based Adaptation on the TAS Exemplar”. In: *Proceedings of the 1st International Workshop on Control Theory for Software Engineering*. CTSE 2015. ACM, 2015, pp. 1–8. ISBN: 978-1-4503-3814-1. DOI: 10.1145/2804337.2804338.

[171] S. Shevtsov and D. Weyns. “Keep It SIMPLEX: Satisfying Multiple Goals with Guarantees in Control-based Self-adaptive Systems”. In: *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE’16)*. 2016, pp. 229–241.

[172] S. Shevtsov et al. “Systematic Literature Review on Control-Theoretical Software Adaptation”. In: *IEEE Transactions on Software Engineering* (2017).

[173] A. Solomon et al. “Business Process Adaptation on a Tracked Simulation Model”. In: *Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research*. CASCON ’10. IBM Corp., 2010, pp. 184–198. DOI: 10.1145/1923947.1923967.

[174] G. Sousa, W. Rudametkin, and L. Duchien. “Extending Dynamic Software Product Lines with Temporal Constraints”. In: *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS 17. ACM, 2017.

[175] V. E. Souza et al. “Requirements-driven Software Evolution”. In: *Comput. Sci.* 28.4 (Nov. 2013), pp. 311–329. ISSN: 1865-2034. DOI: 10.1007/s00450-012-0232-2.

[176] M. Spielmann. “Abstract State Machines: Verification Problems and Complexity”. PhD thesis. Bibliothek der RWTH Aachen, 2000.

[177] D. Spinellis. “Notable Design Patterns for Domain Specific Languages”. In: *Journal of Systems and Software* 56.1 (Feb. 2001), pp. 91–99. ISSN: 0164-1212. DOI: 10.1016/S0164-1212(00)00089-3.

[178] J. Spriggs. *GSN – The Goal Structuring Notation. A Structured Approach to Presenting Arguments*. Springer, 2012.

[179] S. D. Stoller et al. “Runtime Verification with State Estimation”. In: *Proceedings of the Second International Conference on Runtime Verification*. RV’11. Springer, 2012, pp. 193–207. ISBN: 978-3-642-29859-2. DOI: 10.1007/978-3-642-29860-8_15.

[180] G. Su et al. “An Iterative Decision-Making Scheme for Markov Decision Processes and Its Application to Self-adaptive Systems”. In: *19th International Conference on Fundamental Approaches to Software Engineering (FASE)*. 2016, pp. 269–286.

[181] J. Swanson et al. “Beyond the Rainbow: Self-adaptive Failure Avoidance in Configurable Systems”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. ACM, 2014, pp. 377–388. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635915.

[182] D. Sykes, J. Magee, and J. Kramer. “FlashMob: Distributed Adaptive Self-assembly”. In: *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS ’11. ACM, 2011, pp. 100–109. ISBN: 978-1-4503-0575-4. DOI: 10.1145/1988008.1988023.

[183] P. Tabuada. *Verification and Control of Hybrid Systems*. Springer, 2009.

[184] B. Takhedmit and K. Abbas. “A parametric uncertainty analysis method for queues with vacations”. In: *Journal of Computational and Applied Mathematics* 312.Supplement C (2017). ICMCMST 2015, pp. 143–155. ISSN: 0377-0427. DOI: https://doi.org/10.1016/j.cam.2016.02.031.

[185] G. Tamura et al. “Towards Practical Runtime Verification and Validation of Self-Adaptive Software Systems”. In: *Software Engineering for Self-Adaptive Systems II*. Vol. 7475. Lecture Notes in Computer Science. Springer, 2013.

[186] G. Tamura et al. “Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers”. In: Springer, 2013. Chap. Towards Practical Runtime Verification and Validation of Self-Adaptive Software Systems, pp. 108–132. ISBN: 978-3-642-35813-5. DOI: 10.1007/978-3-642-35813-5_5.

[187] C. Ţăpuş, I.-H. Chung, and J. K. Hollingsworth. “Active Harmony: Towards Automated Performance Tuning”. In: *ACM/IEEE Conference on Supercomputing*. SC ’02. IEEE Computer Society Press, 2002, pp. 1–11.

[188] G. Tesauro et al. “A multi-agent systems approach to autonomic computing”. In: *Third International Conference on Autonomous Agents and Multiagent Systems*. 2004, pp. 464–471.

[189] J. Tretmans. “Formal methods and testing”. In: Springer, 2008. Chap. Model based testing with labelled transition systems, pp. 1–38. ISBN: 3-540-78916-2, 978-3-540-78916-1.

[190] S. Tschirner, L. Xuedong, and W. Yi. “Model-based Validation of QoS Properties of Biomedical Sensor Networks”. In: *Proceedings of the 8th ACM International Conference on Embedded Software*. EMSOFT ’08. ACM, 2008.

[191] UK Civil Aviation Authority. *Unmanned Aircraft System Operations in UK Airspace — Guidance. CAP 722. Sixth Edition*. 2015.

[192] UK Health & Safety Commission. *The use of computers in safety-critical applications*. 1998.

[193] UK Ministry of Defence. *Defence Standard 00-56, Issue 4: Safety Management Requirements for Defence Systems*. June 2007.

[194] UK Office for Nuclear Regulation. *The Purpose, Scope, and Content of Safety Cases, Rev. 3*. July 2013.

[195] University of Virginia Dependability and Security Research Group. *Safety case repository*. 2014.

[196] US Dept. Health and Human Services, Food and Drug Administration. *Infusion Pumps Total Product Life Cycle—Guidance for Industry & FDA Staff*. 2014.

[197] S. Uttamchandani et al. “CHAMELEON: A Self-Evolving, Fully-Adaptive Resource Arbitrator for Storage Systems”. In: *USENIX Annual Technical Conference, General Track*. 2005, pp. 75–88.

[198] A. Valmari. “The state explosion problem”. In: *Lectures on Petri Nets I: Basic Models: Advances in Petri Nets*. Springer, 1998, pp. 429–528. ISBN: 978-3-540-49442-3. DOI: 10.1007/3-540-65306-6_21.

[199] N. M. Villegas et al. “A Framework for Evaluating Quality-driven Self-adaptive Software Systems”. In: *6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. 2011, pp. 80–89.

[200] T. Vogel and H. Giese. “Model-Driven Engineering of Self-Adaptive Software with EUREMA”. In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 8.4 (Jan. 2014), 18:1–18:33. ISSN: 1556-4665. DOI: 10.1145/2555612.

[201] W. Walsh et al. “Utility functions in autonomic systems”. In: *IEEE International Conference on Autonomic Computing*. 2004, pp. 70–77.

[202] Y. Wang, M. Martonosi, and L.-S. Peh. “Predicting Link Quality Using Supervised Learning in Wireless Sensor Networks”. In: *SIGMOBILE Mob. Comput. Commun. Rev.* 11.3 (July 2007), pp. 71–83. ISSN: 1559-1662.

[203] D. Weyns. “Software Engineering of Self-Adaptive Systems: An Organised Tour and Future Challenges”. In: *Chapter in Handbook of Software Engineering* (2017). (forthcoming; https://people.cs.kuleuven.be/danny.weyns/papers/2017HSE.pdf).

[204] D. Weyns and R. Calinescu. “Tele Assistance: A Self-Adaptive Service-Based System Exemplar”. In: *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. May 2015, pp. 88–92. DOI: 10.1109/SEAMS.2015.27.

[205] D. Weyns and M. U. Iftikhar. “ActivFORMS: An Efficient Approach to Engineer Self-Adaptive Systems with Guarantees”. In: *Under review* (2017).

[206] D. Weyns, S. S., and S. Pllana. “Providing Assurances for Self-Adaptation in a Mobile Digital Storytelling Application Using ActivFORMS”. In: Self-Organising and Self-Adaptive Systems SASO. 2014.

[207] D. Weyns. “Towards an Integrated Approach for Validating Qualities of Self-adaptive Systems”. In: *Proceedings of the Ninth International Workshop on Dynamic Analysis*. ACM, 2012, pp. 24–29. ISBN: 978-1-4503-1455-8.

[208] D. Weyns and T. Ahmad. “Claims and Evidence for Architecture-based Self-adaptation: A Systematic Literature Review”. In: *Proceedings of the 7th European Conference on Software Architecture*. ECSA’13. Springer, 2013, pp. 249–265. ISBN: 978-3-642-39030-2. DOI: 10.1007/978-3-642-39031-9_22.

[209] D. Weyns and M. U. Iftikhar. “Model-based Simulation at Runtime for Self-adaptive Systems, 2017”. In: *Models at Runtime, IEEE International Conference on Autonomic Computing (ICAC)*. MODELS 2016. © 2016 IEEE. Reprinted, with permission.

[210] D. Weyns, M. U. Iftikhar, and J. Söderlund. “Do External Feedback Loops Improve the Design of Self-adaptive Systems? A Controlled Experiment”. In: *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS ’13. IEEE, 2013, pp. 3–12. ISBN: 978-1-4673-4401-2.

[211] D. Weyns, S. Malek, and J. Andersson. “FORMS: Unifying reference model for formal specification of distributed self-adaptive systems”. In: *ACM Transactions on Autonomous and Adaptive Systems* 7.1 (2012), p. 8.

[212] D. Weyns et al. “Claims and Supporting Evidence for Self-adaptive Systems: A Literature Study”. In: *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS ’12. IEEE, 2012, pp. 89–98. ISBN: 978-1-4673-1787-0.

[213] D. Weyns et al. “Perpetual Assurances in Self-Adaptive Systems”. In: *Software Engineering for Self-Adaptive Systems III*. Lecture Notes in Computer Science, Springer, 2016.

[214] D. Weyns et al. “A Survey of Formal Methods in Self-adaptive Systems”. In: *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*. C3S2E ’12. ACM, 2012, pp. 67–79. ISBN: 978-1-4503-1084-0. DOI: 10.1145/2347583.2347592.

[215] S. R. White et al. “An architectural approach to autonomic computing”. In: *International Conference on Autonomic Computing*. IEEE. 2004, pp. 2–9.

[216] J. Whittle et al. “RELAX: a language to address uncertainty in self-adaptive systems requirement”. In: *Requirements Engineering* 15.2 (2010). ISSN: 0947-3602. DOI: 10.1007/s00766-010-0101-0.

[217] J. Whittle et al. “RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems”. In: *17th IEEE International Requirements Engineering Conference*. Aug. 2009, pp. 79–88. DOI: 10.1109/RE.2009.36.

[218] L. D. Xu, W. He, and S. Li. “Internet of Things in Industries: A Survey”. In: *IEEE Transactions on Industrial Informatics* 10.4 (Nov. 2014), pp. 2233–2243. ISSN: 1551-3203. DOI: 10.1109/TII.2014.2300753.

[219] H. L. S. Younes. “Verification and Planning for Stochastic Processes with Asynchronous Events”. AAI3159989. PhD thesis. 2004. ISBN: 0-496-93475-9.

[220] M. Zamani, N. van de Wouw, and R. Majumdar. “Backstepping controller synthesis and characterizations of incremental stability”. In: *Systems & Control Letters* 62.10 (2013), pp. 949–962. DOI: 10.1016/j.sysconle.2013.07.002.

[221] J. Zhang and B. H. Cheng. “Using temporal logic to specify adaptive program semantics”. In: *Journal of Systems and Software* 79.10 (2006), pp. 1361–1369. ISSN: 0164-1212. DOI: http://dx.doi.org/10.1016/j.jss.2006.02.062.

[222] J. Zhang, H. J. Goldsby, and B. H. Cheng. “Modular Verification of Dynamically Adaptive Systems”. In: AOSD. 2009.

[223] G. Zhou et al. “Impact of Radio Irregularity on Wireless Sensor Networks”. In: *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services*. MobiSys ’04. ACM, 2004. ISBN: 1-58113-793-1.

[224] P. Zoghi et al. “Designing Adaptive Applications Deployed on Cloud Environments”. In: *ACM Transactions on Autonomous and Adaptive Systems* 10.4 (Jan. 2016), 25:1–25:26.

ISBN: 978-91-88761-04-0 (print), 978-91-88761-05-7 (pdf)

Linnéuniversitetet

KU LEUVEN